FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

PolyCoDif

A Method for Semantic Patches of Multiple Programming Languages Based on Continuously Captured Changes

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Business Informatics

eingereicht von

Matthias Sperl
Matrikelnummer 00925873

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig
Mitwirkung: Johann Grabner

Wien, 25. September 2019 ____________________ ____________________

(Unterschrift Verfasser/In) (Unterschrift Betreuung)

PolyCoDif

A Method for Semantic Patches of Multiple Programming Languages Based on Continuously Captured Changes

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Business Informatics

by

Matthias Sperl
Registration Number 00925873

elaborated at the
Institute of Information Systems Engineering
Research Group for Industrial Software
to the Faculty of Informatics
at TU Wien

Advisor: Thomas Grechenig
Assistance: Johann Grabner

Vienna, September 25, 2019

Technische Universität Wien, Forschungsgruppe INSO
A-1040 Wien ▪ Wiedner Hauptstr. 76/2/2 ▪ Tel. +43-1-587 21 97 ▪ www.inso.tuwien.ac.at

Statement by Author

Matthias Sperl
Wulzendorfstraße 24a/5, 1220 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

I hereby declare that I am the sole author of this thesis, that I have completely indicated all sources and help used, and that all parts of this work – including tables, maps and figures – if taken from other works or from the internet, whether copied literally or by sense, have been labelled including a citation of the source.

(Place, Date) (Signature of Author)

Kurzfassung

Moderne Werkzeuge der Softwareentwicklung bieten eine Vielzahl an Möglichkeiten Softwareentwickler*innen in ihrer täglichen Arbeit zu unterstützen. Diese Werkzeuge bieten nicht nur einfache Funktionen, basierend auf den Worten in einer Datei, sondern analysieren die Struktur des Quellcodes. Besonders Operationen zur automatisierten Überarbeitung des Quelltextes werden zunehmend raffinierter und erleichtern die Arbeit von unzähligen Entwickler*innen beim Schreiben von neuem Code. Ein Beispiel dafür ist die Operation *Signatur ändern* der integrierten Entwicklungsumgebung *IntelliJ IDEA*. Diese Fähigkeit ermöglicht es Entwickler*innen Parameter zu Funktionen hinzuzufügen oder zu entfernen. Das Programm versucht dabei möglichst intelligent diese Parameter auf der Aufruferseite zu vervollständigen.

Im Gegensatz dazu operiert Software zur Überprüfung und zum Review von Code noch immer auf der textuellen Repräsentation. Der Unterschied zwischen zwei Versionen wird Zeile für Zeile, Wort für Wort, Buchstabe für Buchstabe berechnet, ohne die Bedeutung oder die baumartige Struktur von Code zur Hilfe zu nehmen. Würde semantische Information miteinbezogen werden, könnten stilistische Änderungen, die keine Auswirkungen auf die Laufzeit haben, als irrelevant markiert werden.

Das Ziel dieser Arbeit ist es, existierende Algorithmen zur Berechnung der Differenz zwischen zwei Versionen von Quellcode so zu erweitern, dass diese mit mehreren Programmiersprachen umgehen können.

Zusätzlich werden derzeit Änderungen am Quelltext mit Hilfe von expliziten *commits* nachvollzogen. Das führt dazu, dass die Historie zwischen diesen Zeitpunkten verloren geht. Daher werden im Laufe dieser Arbeit diese Algorithmen erweitert damit sie auch auf die kontinuierlichen Änderungen zugreifen können, die sich während der Entwicklung von Software ergeben. Diese kontinuierliche Verfolgung soll einen detaillierteren Einblick in die Evolution der Software ermöglichen.

Die praktische Anwendbarkeit dieses Prototyps wird in einer qualitativen Benutzerstudie getestet. Die Teilnehmer*innen an der Studie müssen Fragen zu vier Szenarien beantworten und die Benutzerfreundlichkeit des Programmes auf einer standardisierten Skala bewerten. Der Prototyp erreichte einen Wert von 73,75 auf der *System Usability Scale*.

Schlüsselwörter

Quellcode Differenz Extraktion, Algorithmen für Änderungserkennung von Baumstrukturen, kontinuierliche Änderungserkennung, Software Evolution

Abstract

Modern tools for developing software have added a plethora of capabilities to support software engineers in their daily work. These tools do not only provide simple features based on the words in a file, but parse and understand the code, and extract the underlying structure. Especially operations for refactoring have become more and more sophisticated. It is possible with a couple of clicks to change the signature of methods, move classes, extract variables, and structurally search and replace code snippets. These advances in writing new code and adapting existing code have been a major advantage for developers.

While integrated development environments and editors like IntelliJ IDEA can perform more and more operations based on the structure of the source code, tools for reviewing code are still stuck with the textual representation of the code. The difference between two versions of the source code is calculated line by line, word by word, character by character without any regard for the semantic meaning and tree-like structure of source code. Diff algorithms which work on the abstract syntax trees can, for example, mark changes in the code which have no effect on the runtime behavior of the program. If, for example, a programmer changes the formatting of the code, this will result in many detected changes when using classic text-based algorithms. Semantic diff algorithms could hide these changes so that a reviewer can focus on semantically relevant changes.

The goal of this thesis is to take existing algorithms for calculating the difference between two versions of source code in a tree structure and extend them to work with multiple programming languages.

Current Version Control Systems are tracking the changes only on actions of software engineers to track change between certain points in the history. This results in a loss of information between these points in time. To remedy that, the semantic diff algorithms are combined with approaches to continuously track changes of source code without relying on the developer to record fixed points in time with explicit commits.

A qualitative user study tests the usefulness of the resulting prototype, where developers try to answer questions about source code in four different scenarios. The usability of the tool is judged by the participants of the user study based on the *System Usability Scale* and reached an overall value of 73,75.

Keywords

Source code change extraction, tree-differencing algorithms, continuous tracking, polyglot, software repositories, software evolution analysis.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

List of Abbreviations

ADT	algebraic data type
API	application programming interface
AST	abstract syntax tree
CMS	Content Management System
GUI	Graphical User Interface
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
LSP	Language Server Protocol
PSI	Program Structure Interface
REST	Representational State Transfer
SUS	System Usability Scale
UI	User Interface
VCS	Version Control System

1 Introduction

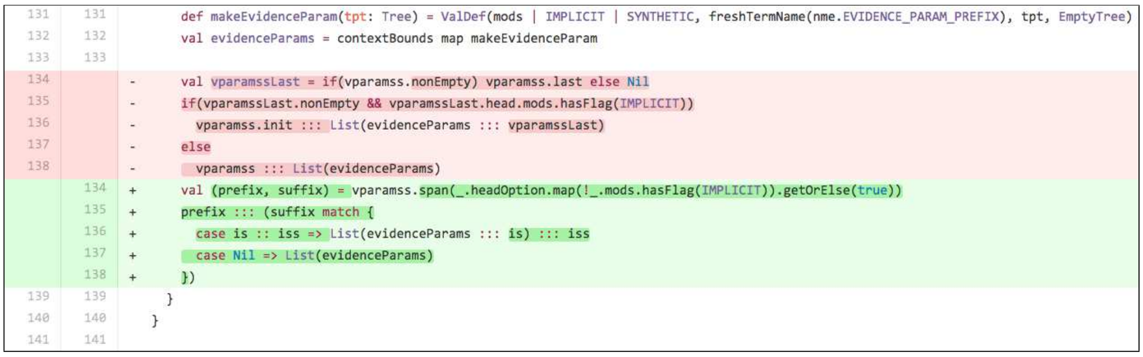
The goal of this thesis is to advance the state of the art how software engineers find information about the history of the projects they are working on. To make informed decisions of the future of source code it is always beneficial to understand its past [21]. This introduction elaborates on the points why a continuous and semantic approach is beneficial, how this thesis tries to proof this points and how it is structured.

1.1 Motivation

Version Control Systems (VCSs) have become a standard tool for tracking changes within source code [53] and have a demonstrable and significant impact on developer productivity [3]. VCS like *Git* [23] or *Subversion* [1] operate only on text level and have no inherent understanding of the syntax of the text. Therefore, when comparing two different points in time, these tools present the difference solely on a textual basis.

Figure1.1 is an example of this behavior. The red background shows the removal of text and the green text represents all the additions. The sample diff is taken from a pull request against the Scala compiler [48]. The proposed improvement is only four lines and can be split into two semantic changes. The first one in line 134 adapts how the `vparams` list is split up. Previously the algorithm used the last element of the collection, but now the list is separated by a predicate that checks for an `IMPLICIT` flag. The second change rewrites the `if` to a pattern match. After analyzing the difference, a developer should know, that the `else` and the `case Nil` branch of the change have the same semantic behavior, but this is by no means trivial without intimate knowledge of the code. The diff, provided by GitHub, does not assist the developers to recognize the semantics of the textual changes.

However, current research provides algorithms to compare source code structurally. Instead of comparing the code line by line these algorithms are based on graphs. One common representation of source code in graph form is an abstract syntax tree. Figure 1.2 shows a visualization of such a tree structure. The advantages of an abstract syntax tree is that it „captures the essential structure of the input in a tree form, while omitting unnecessary syntactic details“ [31] and is, therefore, useful for code analysis because it provides additional information to the textual representation of



```
131  131       def makeEvidenceParam(tpt: Tree) = ValDef(mods | IMPLICIT | SYNTHETIC, freshTermName(nme.EVIDENCE_PARAM_PREFIX), tpt, EmptyTree)
132  132       val evidenceParams = contextBounds map makeEvidenceParam
133  133
134       -    val vparamssLast = if(vparamss.nonEmpty) vparamss.last else Nil
135       -    if(vparamssLast.nonEmpty && vparamssLast.head.mods.hasFlag(IMPLICIT))
136       -      vparamss.init ::: List(evidenceParams ::: vparamssLast)
137       -    else
138       -      vparamss ::: List(evidenceParams)
     134  +    val (prefix, suffix) = vparamss.span(_.headOption.map(!_.mods.hasFlag(IMPLICIT)).getOrElse(true))
     135  +    prefix ::: (suffix match {
     136  +      case is :: iss => List(evidenceParams ::: is) ::: iss
     137  +      case Nil => List(evidenceParams)
     138  +    })
139  139     }
140  140   }
141  141
```

Figure 1.1: An example of a textual diff generated via GitHub

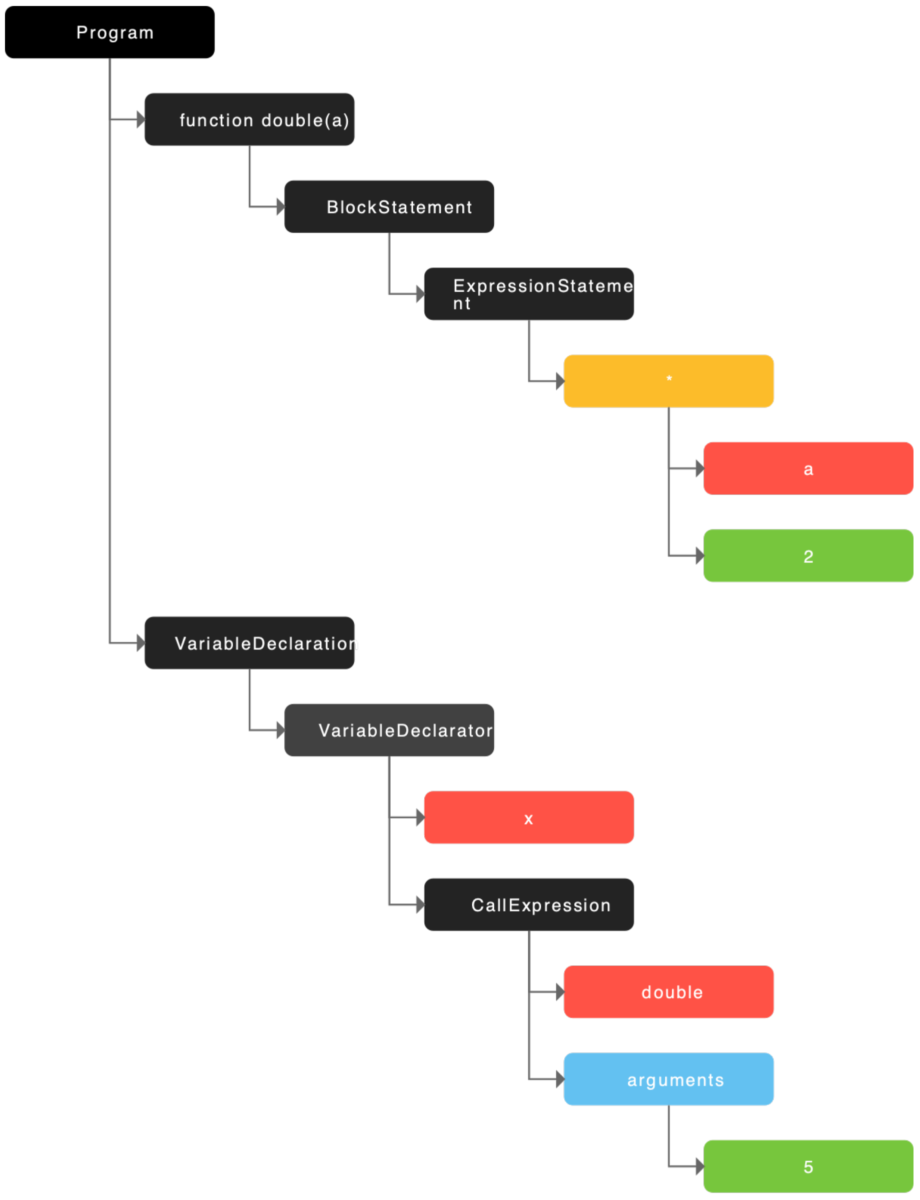


Figure 1.2: A graphical visualization for the code snippet `def double(a){ a*2; };var x = double(5);`

the code and omits superfluous information, e.g. white spaces, statement delimiters as described by Kawrykow and Robillard [33].

Even though comparing the structure of one programming language is useful, with the advent of polyglot programming [18] it is necessary that difference tools can deal with multiple languages. These multi-language capabilities are especially convenient when rewriting parts of an application to use a different programming language or when developers use DSLs within another language. In the case of using one language embedded within another language, it is necessary that the tool can track the location of the embedded code.

Tracking semantic differences requires a fine granular unit of change to efficiently recognize the operations performed by a programmer on the source code. Controlling the size of the change-set so that it provides enough information for the semantic differencing without collecting too much unnecessary information and providing a fast feedback for the user is crucial for a useful next generation differencing tool.

Additionally, it has been shown that more than a third of all code changes do not reach the VCS in the form of a commit [42], and there are several discussions about the ideal commit size [51]. Doing all these operations is very tedious and requires a high cognitive load. An automated differencing tool should relief the developer of this burden.

1.2 Research Questions

Keeping the motivation of the thesis in mind, the following research questions were posed:

- How to combine a tree differencing algorithm with continuous change tracking on top of a polyglot abstract syntax tree?
- What advantages provides a difference viewer based on a continuous, AST change detection for developers?
- How do developers rate the usability of a difference viewer based on continuous AST change detection?

1.3 Aim of the Work

The expected outcome of this thesis is a prototype of an improved diff tool based on a generalized framework for comparing semantic differences, a test plan to evaluate and compare the prototype against existing tools, and the evaluation of the prototype's usefulness according to the test plan. In contrast to current tools, the change tracking will be done continuously to counter missing information that was described in section 1.1. The difference between two versions of a file will be presented based on the semantic meaning of the source code. To achieve this, the application will create abstract syntax trees for the two different versions of the source code file and implement a differencing algorithm based on these two graphs. A unified semantic representation of the source code will be the key to enable efficient comparisons with multiple languages. For a first version PolyCoDif will focus on comparing source code written in the same programming languages with each other. This means, that PolyCoDif should work with e.g. projects with Scala and Java source code, but for now it will not compare Java files with Scala files. In the future the tool could be extended to support comparing different programming languages to one another quite easily.

These cross-language capabilities of the developed prototype will be one of the features that sets „PolyCoDif“ apart from other differencing tools. However, the unique contribution of this tool is the combination of continuous tracking, semantic differencing and the multi-language capabilities in one application.

As a byproduct, a generalized framework for comparing the semantic difference between versions of source code across different programming languages on multiple platforms needs to be developed. To this end, a user interface will be designed. As a proof of the cross-language capabilities, a back-end for Java and Scala will be implemented.

To reach the expected results, a system comprised of four components will be designed and implemented. These four components are:

- A generic system which is capable of continuously tracking changes made to source code.
  - An implementation of the tracker.

- A generic structural code difference analyzer
  - for Scala.
  - for Java.
- A difference viewer.
- A storage back-end built upon git.

1.4 Structure of the Work

The thesis is split into three basic parts. In the chapter state of the art a systematic review of the literature is done. Based on the existing literature, the most relevant tools are identified and compared.

The tools and algorithms found in the literature and used in the industry are used as a basis to implement a prototype for the tool. This implementation is described in the Chapter 4 and is split into several sections. The first section is about the overall *architecture* of the solution, explaining the different modules and their purposes. Afterwards the identified *semantic operations* are developed. The *visualization* of these operations is the focus of the following chapter. Section *Unified abstract syntax tree* illustrates how these operations work in a multi-language setting. The core difference algorithms are developed in a section for the *Tree-based diff* and the *Text-based diff*. Finally, in Section *continuous refinement of the patch-set* these algorithms are enhanced to work with fine-grained, continuously gathered data changes.

The development of the prototype followed the same iterative procedure which was outlined above. It started out with the design and implementation of the semantic operations and possible visualizations of these operations. To work with multiple programming languages a unified abstract syntax tree is defined in the next section. Next the different possibilities to calculate edit-scripts are explored. Then these algorithms are expanded and adopted to deal with continuous changes.

To evaluate the prototype a scenario-based expert evaluation is conducted. The study is described in Section 5.1 and discussed in the following sections. The chapter has three sections and describes the *scenario-based expert evaluation*, followed by the *results* of the evaluation and recognizes *threats to the validity* of these results.

The discussions chapter attempts to answer the research questions posed in previously with the help of the implementation and evaluation chapters. Finally, the summary and outlook of potential future research opportunities is given.

2 State of the Art

Version control systems for source code have been implemented and used since 1975 with the paper by Rochkind [46]. One of the first widely used version control systems was RCS [54].

While these first tools already implemented a lot of the ideas prevalent in today's systems, advances in diffing algorithms made most of them obsolete. For example, the Myers Algorithm [40] can be found in most modern version control systems. This and other improvements led to the most commonly used version control systems, namely SVN, Git and Mercurial. Additionally, research was focused on putting version control systems and their operations on a more principled theory.

Independently from these advances in version control systems, diff algorithms for trees were developed. For example, by Shasha and Zhang [50]. The initial focus was on documents with an underlying tree structure, e.g. XML.

In this chapter, a systematic literature review is presented, starting with a short history of VCSs cumulating in the theory of patches by Mimram and Di Giusto [39].

Because PolyCoDif combines *continuous tracking* with *semantic patches* in a *multi-language setting*, a survey of the respective state of the art is given in Sections 2.2, 2.4, and 2.5.

There has been scientific research to combine continuous tracking with semantic patches (described in Section 2.6) and to do semantic patches in a programming language agnostic way (described in Section 2.7).

2.1 Theoretical Foundations

While many types of software have version control integrated, like Content Management System (CMS) or word processors, this thesis focuses on version control for source code. Most VCS have a shared terminology to introduce the basic concepts of version control. For this thesis we will use the definitions which are common to both SVN and Git and are described by Pilato, Collins-Sussman, and Fitzpatrick [44] and Chacon [11].

Diff A single change between two files in the source code.

Patch A list of *diffs*, describing one atomic change to the codebase. Although it does sound counter-intuitive the list of diffs might be empty.

Commit A *patch* combined with metadata, e.g. a message, date, and author and optionally a reference to its parent commits.

The above introduced concepts are all in a relation to each other. Each commit has zero or more children. No child means that this represents a (possibly temporary) end state of the development. One child means a linear development, e.g. if a single developer works on the system. Multiple children mean that multiple developers are working on the same repository in parallel.

The situation is similar for the parents of a commit. Each commit can have again zero or more parents. Zero parents are only possible for the initial commit in a new repository, while one parent represents a linear development. The most interesting case is when a commit has multiple parents. This happens if multiple developers integrate their commits to a single new version. Frequently called a **merge**. Besides the multiple parents, a merge commit is nothing special. However, since the multiple parents might have introduced conflicting changes, the patch of the merge commit can be used to resolve this conflict. If no conflict resolution is necessary, a merge commit just contains an empty patch. [11, 44]

In recent years there was an effort to develop a more principled theory behind patches and version management with one example being the software *Darcs* by Roundy [47]. While SVN and Git focus and store snapshots of data for different versions and only compute the diff on demand, Darcs focuses on changes and patch-sets. This makes it possible to freely rearrange the history, combine different patches and apply patches in different order. A cherry pick also never creates a new commit identity but applies the exactly same commit to the history. The theoretical foundations of the patch theory used by Darcs were given by Jacobson [29] and enable most of the optimizations. The same principles can also be used to ease the implementation of continuous tracking and is therefore exploited in the prototype and explained in more detail in Section 4.

2.2 Continuous Tracking

Although continuous tracking of fine-grained changes in a software repository is a relatively young field, software evolution was researched starting in the 90s according to Robbes [45]. Commits are rarely granular enough to give a detailed view of the code's evolution.

There are two approaches of continuous tracking which are elaborated in the next sections.

2.2.1 Evolution Reconstruction

This approach tries to reconstruct the fine-grained changes by reconstructing the operations. An example of this is the identification of refactoring from source code changes by Weissgerber and Diehl [58]. The focus of this research is to automatically resolve conflicts which arise of refactoring the source code. The paper identifies a number of operations divided into **Structural Refactoring** and **Local Refactoring**. In detail, the operations detected on a structural level are:

- Move Class
- Move Interface
- Move Field
- Move Method
- Rename Class

And the operations on a local level are:

- Rename Method
- Hide Method
- Unhide Method
- Add Parameter
- Remove Parameter

As one can deduct from the existence of separate local operations for *Move Field* and *Move Method* but also for the global operations *Move Class* and *Move Interface*, this research is focused on a Java style programming language and does not support more functional constructs like `typeclasses` or algebraic data types (ADTs). These concepts are important to support polymorphism in languages like Haskell as explained by Hudak et al. [28]. Therefore, this is only useful in the context of a single, object-oriented language and cannot be used for a polyglot use-case.

The focus on imperative and object-oriented programming languages can be seen when reviewing the relevant literature. Additional examples of similar algorithms have been developed by Apiwattanapong, Orso, and Harrold [2] and Lahiri et al. [34] or are sold as commercial software under the name semantic merge [52].

2.2.2 Change Monitoring

Instead of doing a potentially expensive calculation to reconstruct the evolution of source code, modern editors and Integrated Development Environments (IDEs)s can be used to track each operation the developer performs.

As one can imagine this can lead to many operations, which are irrelevant to the final state of a commit. Nevertheless all these modifications need to be stored on disk, to be useful for reconstructing the evolution of software [26].

One of these systems to continuously track changes on source code is called Syde and was developed by Hattori [24]. Syde tracks the changes based on the AST of a programming language but is also quite useful in just tracking the operations on the source code. The AST-based tracking is discussed in more detail in Section 2.6.

The concept of Syde was focused on the collaboration of multiple developers on the same source repository and instead of reusing the repository for storing the continuous data gathered on the operation performed on the source code, a separate server-side component was developed. A high-level schematic how the client-side eclipse plugin communicated with the server can be found in Figure 2.1.

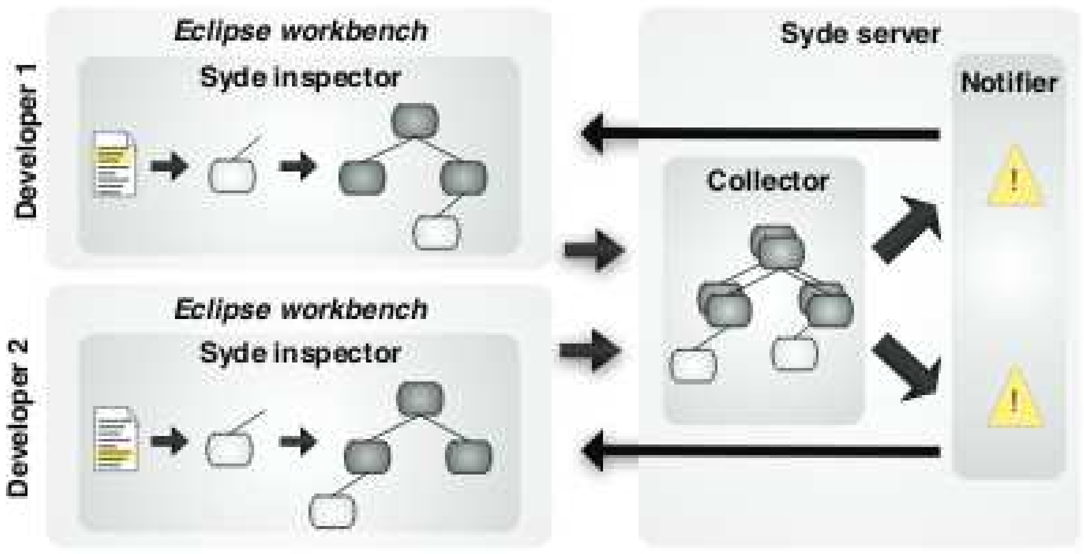
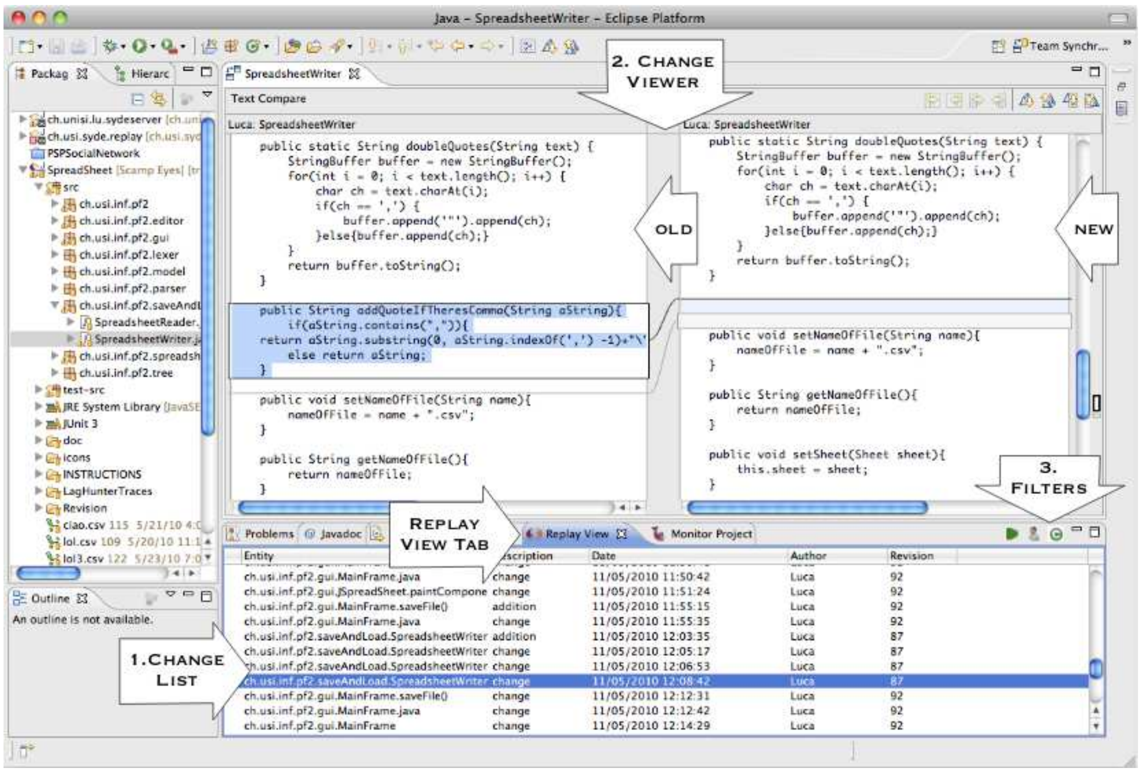


Figure 2.1: The architecture of the Syde server developed in by Hattori [24].

This is then further used to design an UI which makes it easy to track the changes a single developer has done over time. A screenshot of this interface can be found in Figure 2.2.



```
public static String doubleQuotes(String text) {
    StringBuffer buffer = new StringBuffer();
    for(int i = 0; i < text.length(); i++) {
        char ch = text.charAt(i);
        if(ch == ',') {
            buffer.append('"').append(ch);
        }else{buffer.append(ch);}
    }
    return buffer.toString();
}

public String addQuoteIfTheresComma(String aString){
    if(aString.contains(",")){
return aString.substring(0, aString.indexOf(',') -1)+"\'
    else return aString;
}

public void setNameOfFile(String name){
    nameOfFile = name + ".csv";
}

public String getNameOfFile(){
    return nameOfFile;
}
```

```
public static String doubleQuotes(String text) {
    StringBuffer buffer = new StringBuffer();
    for(int i = 0; i < text.length(); i++) {
        char ch = text.charAt(i);
        if(ch == ',') {
            buffer.append('"').append(ch);
        }else{buffer.append(ch);}
    }
    return buffer.toString();
}

public void setNameOfFile(String name){
    nameOfFile = name + ".csv";
}

public String getNameOfFile(){
    return nameOfFile;
}

public void setSheet(Sheet sheet){
    this.sheet = sheet;
}
```

Entity	Description	Date	Author	Revision
ch.usi.inf.pf2.gui.MainFrame.java	change	11/05/2010 11:50:42	Luca	92
ch.usi.inf.pf2.gui.JSpreadSheet.paintCompone	change	11/05/2010 11:51:24	Luca	92
ch.usi.inf.pf2.gui.MainFrame.saveFile()	addition	11/05/2010 11:55:15	Luca	92
ch.usi.inf.pf2.gui.MainFrame.java	change	11/05/2010 11:55:35	Luca	92
ch.usi.inf.pf2.saveAndLoad.SpreadsheetWriter	addition	11/05/2010 12:03:35	Luca	87
ch.usi.inf.pf2.saveAndLoad.SpreadsheetWriter	change	11/05/2010 12:05:17	Luca	87
ch.usi.inf.pf2.saveAndLoad.SpreadsheetWriter	change	11/05/2010 12:06:53	Luca	87
ch.usi.inf.pf2.saveAndLoad.SpreadsheetWriter	change	11/05/2010 12:08:42	Luca	87
ch.usi.inf.pf2.gui.MainFrame.saveFile()	change	11/05/2010 12:12:31	Luca	92
ch.usi.inf.pf2.gui.MainFrame.java	change	11/05/2010 12:12:42	Luca	92
ch.usi.inf.pf2.gui.MainFrame	change	11/05/2010 12:14:29	Luca	92

Figure 2.2: The user interface of Replay developed by Hattori, Lungu, and Lanza [26].

While this can be used to compare different versions and doing reviews on merge requests, the focus of this research is slightly different and more general.

A similar approach was integrated into the LSP [35]. It supports an operation called `DidChange-TextDocument Notification` which is used to send new versions of the document to a server.

2.2.3 Combined Approach

There might be possibilities to combine both approaches to limit the number of changes which need to be stored as part of a commit. However, a literature review has not yielded any significant research in this area.

2.3 Textual Diff

The first studied algorithm to calculate the edit distance between two versions of a program are based on simple tokens. An example for this would be a string-based algorithm by Myers [40]. The author describes several different algorithms based on the same idea.

A graph is created based on the two texts, as can be seen in Figure 2.3. To calculate the edit distance, a way from the top left to the bottom right is needed. A move to the right means deleting a single token, move down means inserting the token. While a diagonal movement means no change at all.

Depending on the algorithm the result is ideally a patch with a *minimal* edit distance. However, to optimize the performance of the calculations, other strategies for calculating the movements can be used, resulting in slightly different patch-sets [6, 40, 41].

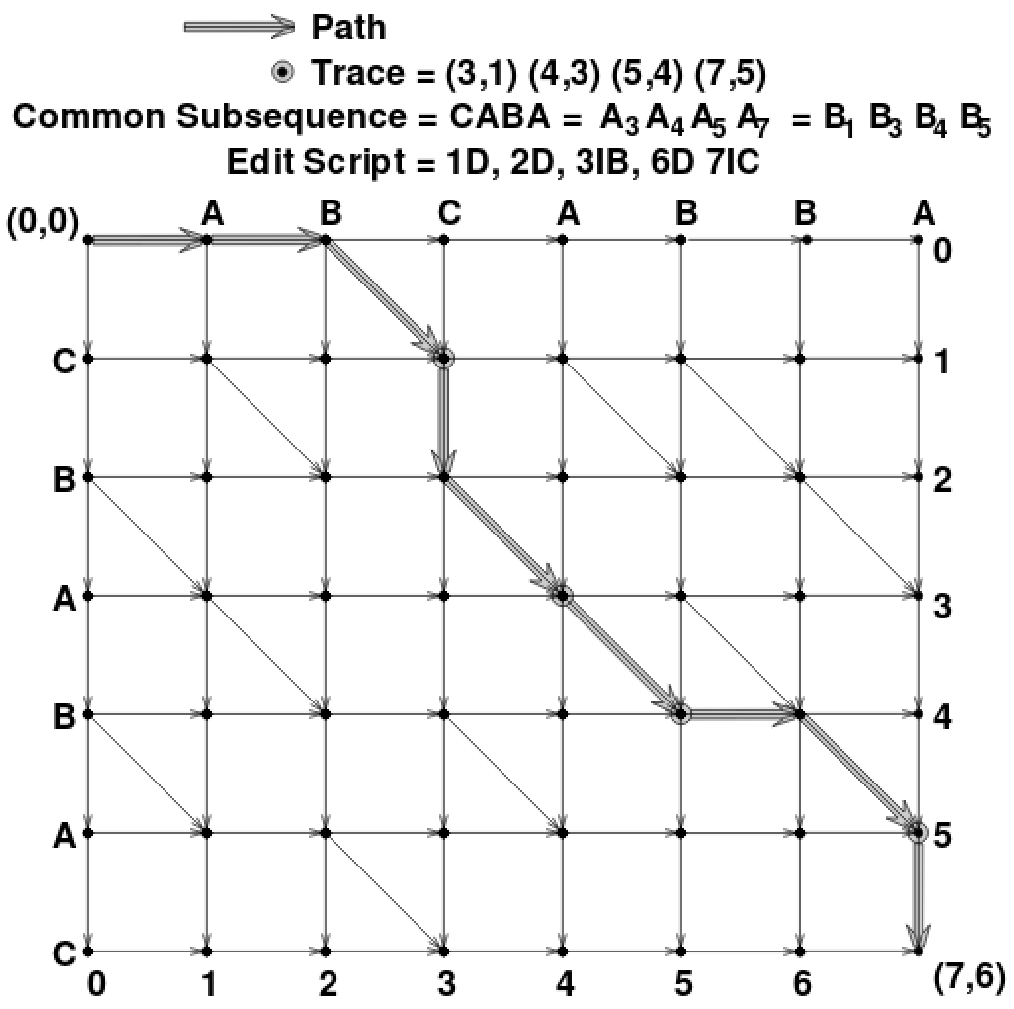

Path
Trace = (3,1) (4,3) (5,4) (7,5)
Common Subsequence = CABA = A3 A4 A5 A7 = B1 B3 B4 B5
Edit Script = 1D, 2D, 3IB, 6D 7IC
(0,0)
A B C A B B A
C B A B A C
0 1 2 3 4 5 6
(7,6)

Figure 2.3: A sample edit graph according to Myers. [40]

2.4 Semantic Structural Diff

Since source code can naturally be expressed as an AST, algorithms for calculating the tree edit distance and a corresponding edit-script can be used to compute the difference between two versions of source code based on the abstract syntax.

2.4.1 Abstract Syntax Tree

ASTs are a way to model the terms and types of programming languages. These trees abstract away semantically irrelevant information like whitespaces or comments. Usually, ASTs are used as the intermediate representation of compilers but are also useful for developing programming language tools. [59]

An example of an AST optimized towards such tools is the scalameta project by Burmako et al. [10], which is used to generate a common data model for developer tools. [9]

Other examples of a tool developed based on an AST is the clone detection developed by Baxter et al. [4] or the code comparison for checking for code plagiarism described by Cui et al.

Tree-based structures are very well understood in computer science and therefore are a good basis for diff tools like PolyCoDif.

The definition for trees is recursive and each element within an AST has a different type which represents a different syntactic element in a source code. A simplified version can be defined as follows.

```
data Ast = ClassDef String [Ast] | FieldDef String (Maybe Ast) |
↪  Constructor Ast | New String Ast | Select Ast String | Var
↪ String Ast | Terminal String
```

With this definition we can represent a programming language which supports, classes with fields and constructors, instantiating these classes, selecting (accessing) fields, assigning expressions to variables, and referencing these variables. As one can see this tree structure can be used to describe a simplified Java-like language but it is ill defined to be used as a basis for e.g. more functional programming languages like Haskell or Scala. There are ways to work around this limitation which are explored in Section 2.5.

2.4.2 Tree Edit Distance

According to Bille [7] there is no common definition of the *Tree edit distance* problem but a good enough definition can be found in his survey paper:

> „Assume that we are given a cost function defined on each edit operation. An edit-script S between T_1 and T_2 is a sequence of edit operations turning T_1 into T_2. The cost of S is the sum of the costs of the operations in S. An optimal edit-script between T_1 and T_2 is an edit-script between T_1 and T_2 of minimum cost and this cost is the tree edit distance. The tree edit distance problem is to compute the edit distance and a corresponding edit-script.“ [7]

To fully explain this definition, an understanding of a minimal set of edit operations is needed. In principle there are only three different kinds of operations needed to express each diff.

Insert

Adds new node to the structure of the tree.

Delete

Removes a node from the structure of the tree.

Relabel

Changes the type of node.

While relabel is a simple operation and only changes the type of a node – e.g. from a value declaration to a method declaration – insert and delete change the structure of the tree. Since

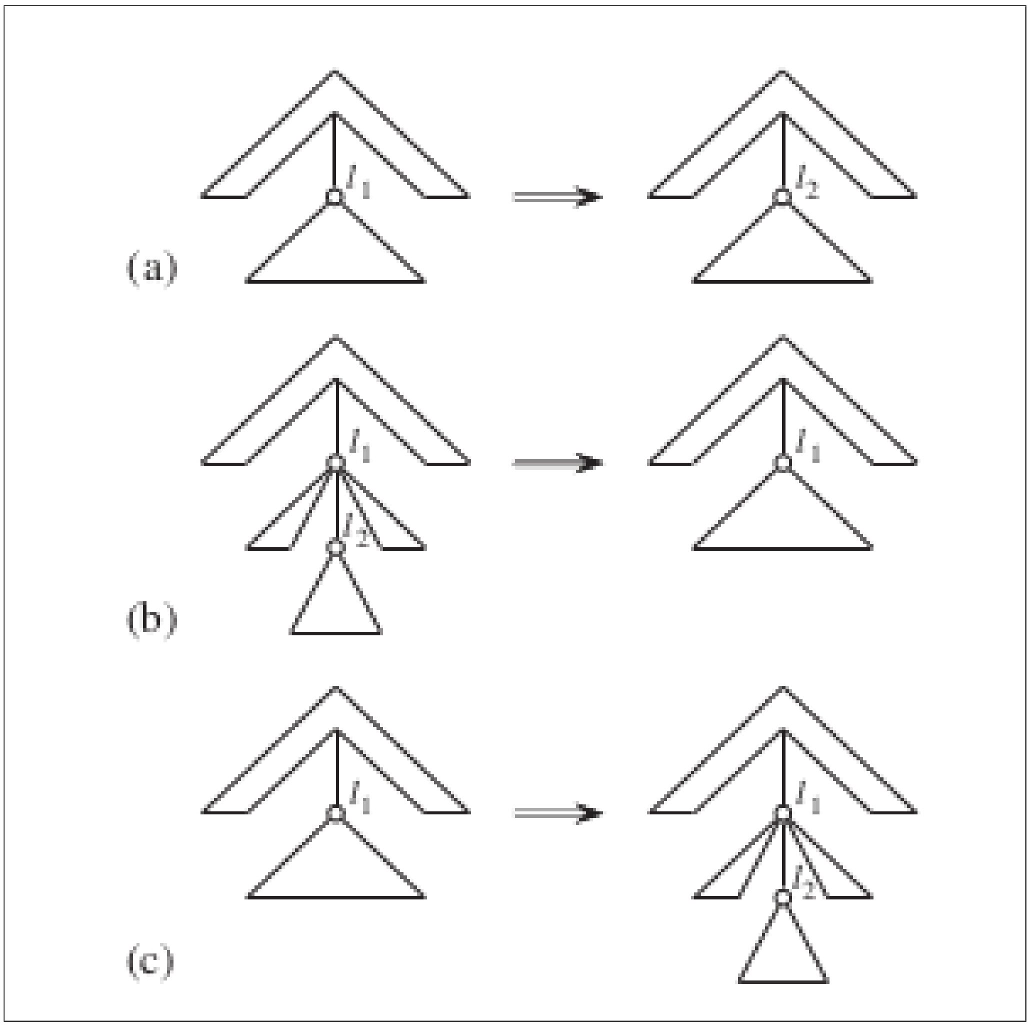


Figure 2.4: (a) A relabeling of the node label l_1 to l_2. (b) Deleting the node labeled l_2. (c) Inserting a node labeled l_2 as the child of the node labeled l_1. [7]

only the specified node is deleted, the algorithm needs to take care to manage the parent and other children accordingly and attach them in the tree without losing the content. A visual explanation given by Bille [7] can be found in Figure 2.4.

In Section 4.5 the edit operations are further refined to better fit the visualization of the diff and the problem domain of source code.

Change Distiller

One of the dominant algorithms found while reviewing the literature was Change Distiller described by Fluri et al. [19]. A description of the algorithm can be found in Figure 2.5.

One of the big advantages of this algorithm is its possibility to customize the specific match functions to accommodate the specifics of different programming languages. Due to the importance and extensibility of this approach, the algorithm is used for *PolyCoDif* and therefore it is explained in detail in Section 4.6.

A quick description of the algorithm can be found in Figure 2.5 with the two extension points $match_1$ and $match_2$.

```
**Input**: trees $T_1$, $T_2$
**Result**: final matching set: $M_{\text{final}}$
$M_{\text{final}} \leftarrow \phi$, $M_{\text{tmp}} \leftarrow \phi$
Mark all nodes in $T_1$ and $T_2$ "unmatched"
**for all** leaf $x \in T_1$ and leaf $y \in T_2$ **do**
    **if** $match_1(x, y)$ **then**
        $M_{\text{tmp}} \leftarrow M_{\text{tmp}} \cup (x, y, sim_{2g}(v(x), v(y)))$
    **end if**
**end for**
Sort $M_{\text{tmp}}$ into descending order, according to the leaf-pair-similarity
**for all** leaf-pair-similarity $(x, y, sim_{2g}(v(x), v(y))) \in M_{\text{tmp}}$ **do**
    $M_{\text{final}} \leftarrow M_{\text{final}} \cup (x, y)$
    Remove all leaf-pairs from $M_{\text{tmp}}$ that contain $x$ or $y$
    Mark $x$ and $y$ "matched"
**end for**
Proceed post-order on trees $T_1$ and $T_2$:
**for all** unmatched node $x \in T_1$, if there is an unmatched node $y \in T_2$ **do**
    **if** $match_2(x, y)$ (incl. dynamic threshold and inner node similarity weighting) **then**
        $M_{\text{final}} \leftarrow M_{\text{final}} \cup (x, y)$
        Mark $x$ and $y$ "matched"
    **end if**
**end for**
```

Figure 2.5: The Change Distiller algorithm according to Fluri et al. [19]

2.4.3 Visualizations of Semantic Edit-scripts

One example of a recent paper in the area of visualizing the edit-scripts generated by algorithms like Change Distiller is the tool *DiffViz* by Frick, Wedenig, and Pinzger [20]. Unfortunately, this research was not yet publicly available when this thesis was started. In future research it might be interesting to integrate *PolyCoDif* with *DiffViz*.

2.5 Multi-language Tooling

With the proliferation of programming languages, it is required to easily extend a specific tool to support a diverse set of paradigms.

As an example, the IDE „IntelliJ IDEA“ uses the concept of a Program Structure Interface (PSI) tree. This specific kind of AST is used to implement most complex transformations, refactoring operations and code highlighting [30]. Since this is a semantic approach, more details are provided in Section 2.7.

An even simpler approach to multi-language tooling is to not do any semantic analysis at all. Even if there are special search algorithms based on semantic information of the code like e.g.

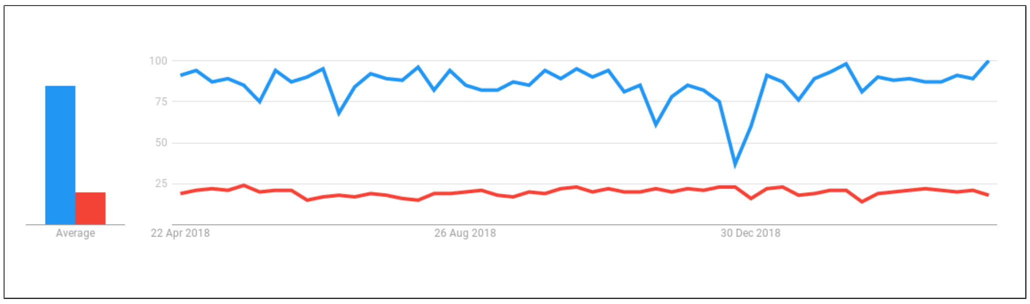


Figure 2.6: The blue color represents `grep`, red represents the popularity of `hoogle`. Generate by Google Trends on the 20th of April 2019.

`hoogle`[1] a simple text-based tool like `grep`[2] is seems to be more widely used by developers. Google Trends[3] was used to do a rough comparison of the popularity of both tools. The Figure 2.6 shows the result of this comparison.

What these simple tool lack in features, they make up in the speed of execution and robustness. Developers of these tools do not need to deal with parsing the source code and therefore don't require the possibility to do error recovery as described by Medeiros and Mascarenhas [37]. Text-based programs work even when the source code is uncompilable or unparsable which happens quite frequently in combination with a continuous change tracking approach. Adding a single character to a source file will often times result in uncompilable text. Fortunately, the implementation of PolyCoDif and the advances in error recovery during parsing by Medeiros and Mascarenhas [36–38] have shown that in most cases it is still possible to parse the source code.

Another example for such a language agnostic tool diff tool is Git. It tracks the differences in files without any semantic information.

2.6 Analysis Continuous Tracking & Semantic Structure Approaches

During the literature review it became apparent that *Syde* was a very thoroughly studied system with a lot of overlap with the planned features for PolyCoDif. This Section, therefore, focuses on the *Syde* system to study a continuous approach for change tracking. According to Hattori and Lanza:

> „Syde provides to the developers of a team the notion of synchronous development, where everyone is aware of the activity of others in real time.“ [25]

The paper not only captures text-based differences but extends to operations which are used during refactoring. The atomic operations it tracks are as following.

Insertion Additions of single characters or sequences of characters.

Deletion Removal of single characters or sequence of characters.

Property Change Adapting the signature of fields and methods of classes. E.g. adding a parameter to a function.

[1] https://hoogle.haskell.org/
[2] https://www.gnu.org/software/grep/
[3] https://trends.google.com/trends/?geo=US

Property Insertion Adding additional fields and methods to classes.

Property Deletion Removing fields and methods from classes.

And the refactoring operations are:

Rename Renaming of a field or method and all its usages.

Move Moving the definition of a field or method within a class without changing the semantics of the class.

First the changes are collected on the client-side and afterwards shipped to a central server. From there the changes are broadcasted to each individual team member. Based on this it visualizes the changes on each developers workstation. The complete system is describe by Hattori and Lanza several papers (see: [24–27]).

2.7 Analysis of Multi-language Capabilities with Semantic Structure

During the literature survey, very little relevant academic work surfaced which is relevant for semantic tooling in regards to a multi-language environment. Most tools either focus on one specific language or use very little semantic information at all.

One prime example of a kind of coding tools which are inherently multi-language are source code editors. These editors support semantic and syntactic highlighting of code based on syntax definitions. Based on these definitions, editors like Sublime can extract indices for efficient and fast *Go To* operations.

One fairly modern such system is the tree-sitter algorithm based on the works of Wagner [56].

> „Tree-sitter is a parser generator tool and an incremental parsing library. It can build a concrete syntax tree for a source file and efficiently update the syntax tree as the source file is edited.“ [56]

More specifically tree-sitter is designed to be „General enough to parse any programming language“ with a robust error recovery. This is essential for development tooling. Most of the time while writing software the code is in a broken state. A single button press rarely results in a fully compilable program. However, it should be possible for developers to use the tools even on these intermediate states [56, 57].

Another example of a very successful multi-language tool which implements features beyond simple *Go To* and code highlighting is the IDE IntelliJ IDEA, mentioned in Section 2.5.

With the PSI structure the platform implements an approach for a unified AST. A unified AST is a tree structure which can be used to represent the internals for ideally any programming language. IntelliJ IDEA uses this data type to implement automatic refactoring only once. However, these refactoring operations are then immediately usable by all language plugins. These parts of the IntelliJ platform are open-source but have not been studied in the academic literature.

2.8 Comparison and Summary of Existing Approaches

While the previous sections of the state of the art talked about the general population of developer tools without focusing too much on code evolution and version tracking, the goal of this section

```
diff --git a/src/reflect/scala/reflect/api/Trees.scala b/src/
↪ reflect/scala/reflect/api/Trees.scala
index 6c498f5e394..355cc65b118 100644
--- a/src/reflect/scala/reflect/api/Trees.scala
+++ b/src/reflect/scala/reflect/api/Trees.scala
@@ -2517,14 +2517,14 @@ trait Trees { self: Universe =>
     *  because pattern matching on abstract types we have here
     ↪ degrades performance.
     *  @group Traversal
     */
-   @deprecated("2.12.3", "Use Tree#traverse instead")
+   @deprecated("Use Tree#traverse instead", "2.12.3")
    protected def itraverse(traverser: Traverser, tree: Tree):
    ↪ Unit = throw new MatchError(tree)

    /** Provides an extension hook for the traversal strategy.
     *  Future-proofs against new node types.
     *  @group Traversal
     */
-   @deprecated("2.12.3", "Use Tree#traverse instead")
+   @deprecated("Use Tree#traverse instead", "2.12.3")
    protected def xtraverse(traverser: Traverser, tree: Tree):
    ↪ Unit = throw new MatchError(tree)

    /** A class that implement a default tree transformation
    ↪ strategy: breadth-first component-wise cloning.
```

Listing 2.1: An example output generated by the `git diff` command

is compare current state of the art tools with regards to code review and visualization of the code history.

2.8.1 Git Integrated Diff

The `git diff` command generates a pure text representation of the change done between two versions. A similar view exists in all modern VCSs and there is also a pure command line tool called `diff` which uses the same visualization.

For example, in Listing 2.1 there is the output of the `git diff` command for a simple change which corrects the order of two arguments used in annotations.

While the representation is straightforward, there is still a lot of information present. For example, the path of the old file and the path of the new file are given in full. Therefore, an implementation of a visualization for a more complex move operation can be added quite easily. In addition, the hashes and therefore the identities of both versions are displayed quite prominently. A very useful information while reviewing the code is the context. Git not only displays the changed line but adds lines before and after each individual change.

Although this representation seems to be quite blunt at first, it has a lot of upsides and should therefore always be the basis for implementing tools to work with the history of code.

One downside of this view is that it is very hard to integrate more information. As for example the UI of GitHub described in the next section shows, one can integrate comments and automated checks as well.

2.8.2 Git-time-machine

Git-time-machine[4] is a plugin for the Atom Editor[5] which gives a visual representation of the history of a repository. An example screenshot from the plugins GitHub project can be seen in Figure 2.7.

The view consists of a *side-by-side* diff and a timeline on the bottom of the screen. According to the author „It shows visual plot of commits to the current file over time and you can click on it on the timeplot or hover over the plot and see all of the commits for a time range.“

2.8.3 GitHub Pull Requests

The main purpose of the GitHub user interface for pull requests is to review changes, before integrating into the main branch of the repository. An example of such a visualization is given in Figure 2.8. The interface is split into four different tabs.

Conversation

This tab displays all comments for specific lines and the overall changes. Because PolyCoDif focuses on reviewing the changes without any possibilities to share the outcome of a review this tab is not that relevant.

Commits

It shows the history of the changes. The number of commits is at least one but especially for larger changes, the pull request might be split up into several chunks, which nevertheless should be merged as one.

Such a view of the history can make the process of reviewing the diff easier if the individual commits are crafted with care. Such a view will be necessary for PolyCoDif as well to give the reviewer a sense of the changes over time.

Checks

The checks interface can be used to display comments created by automated processes like linters or build servers. Even though this can be seen as a kind of automated code review, it is not relevant for a local developer tool.

Files Changed

The changes tab is the most important for this thesis. There are two ways to show the result of the `git diff` command. While the *inline* diff is a graphical version of the `git diff` command, the *side-by-side* is the same diff view as explored in the git-time-machine plugin.

[4] https://github.com/littlebee/git-time-machine
[5] https://atom.io/

2.8.4 GitLab Merge Requests

The GitLab UI is similar to the GitHub visualizations but adds more information about the file tree of the project as can be seen in Figure 2.9. Besides this, the visualization is very similar to the one used by GitHub seen in the previous section.

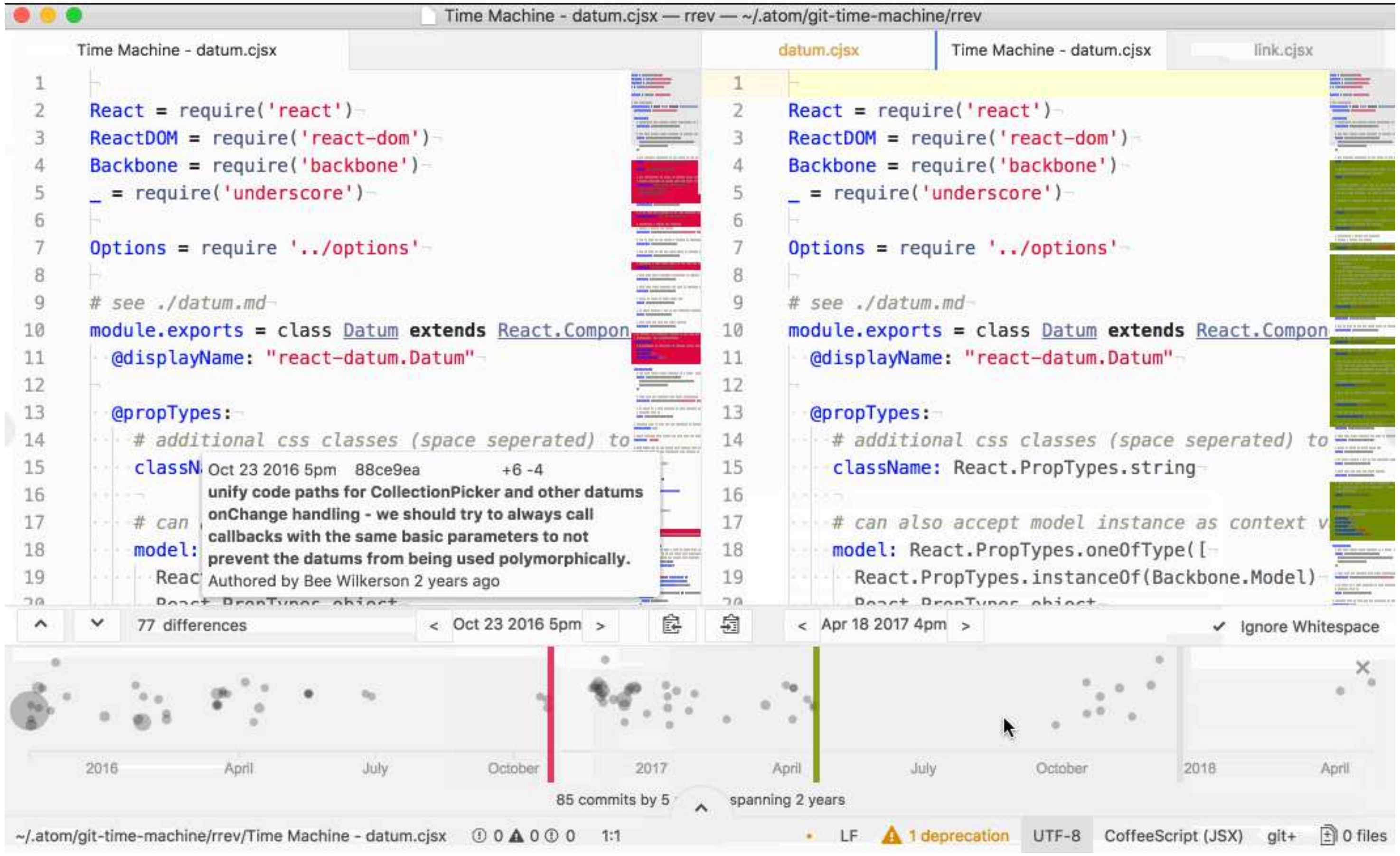
In addition to highlighting the lines which were changed, GitLab uses a darker shaded color to emphasize the individual words which were changed. GitLab recognizes words based on white-spaces.

2.8.5 IntelliJ IDEA Git Integration

IntelliJ has an integrated diff viewer (see Figure 2.10) which defaults to the *side-by-side* diff. It uses different shades of colors to not only highlight the line which was changed but also the tokens which were changed.

Besides, there is a vertical based history view which can be seen in Figure 2.11. This history displays a lot of information in a split view. The left hand side has a table with the summary of the commit message the author and the commit time. To the far left side, there is a graph how the different commits relate to each other.

The right-hand side has more details about each individual commit, including e.g. the files which were touched in the selected commit.



Time Machine - datum.cjsx — rrev — ~/.atom/git-time-machine/rrev

Time Machine - datum.cjsx

```
React = require('react')
ReactDOM = require('react-dom')
Backbone = require('backbone')
_ = require('underscore')

Options = require '../options'

# see ./datum.md
module.exports = class Datum extends React.Compon
  @displayName: "react-datum.Datum"

  @propTypes:
    # additional css classes (space seperated) to
    classN
    
    # can
    model:
      Reac
```

Oct 23 2016 5pm 88ce9ea +6 -4
unify code paths for CollectionPicker and other datums onChange handling - we should try to always call callbacks with the same basic parameters to not prevent the datums from being used polymorphically.
Authored by Bee Wilkerson 2 years ago

datum.cjsx | Time Machine - datum.cjsx | link.cjsx

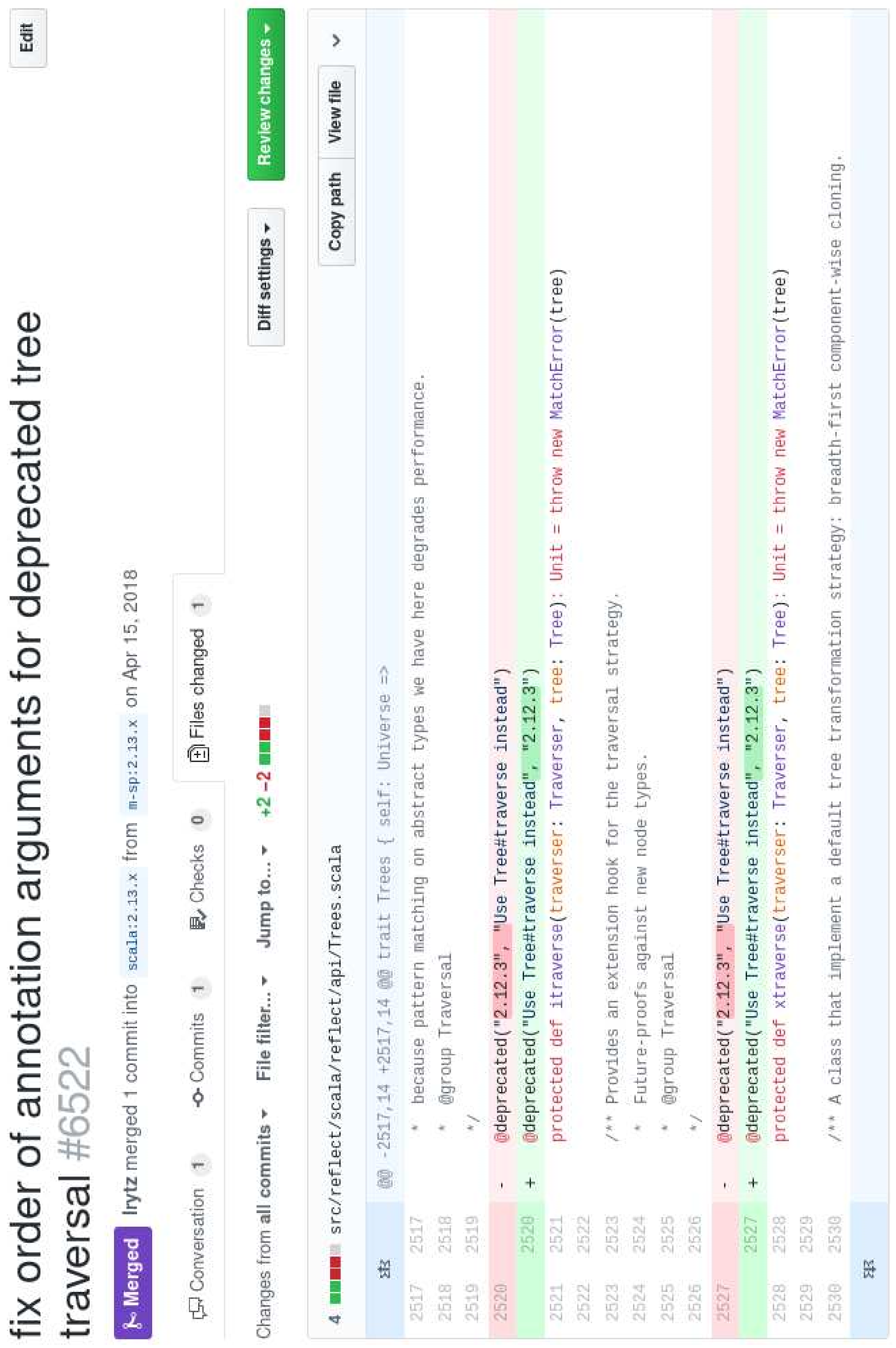
```
React = require('react')
ReactDOM = require('react-dom')
Backbone = require('backbone')
_ = require('underscore')

Options = require '../options'

# see ./datum.md
module.exports = class Datum extends React.Compon
  @displayName: "react-datum.Datum"

  @propTypes:
    # additional css classes (space seperated) to
    className: React.PropTypes.string

    # can also accept model instance as context v
    model: React.PropTypes.oneOfType([
      React.PropTypes.instanceOf(Backbone.Model)
```

77 differences < Oct 23 2016 5pm > < Apr 18 2017 4pm > ✓ Ignore Whitespace

85 commits by 5 spanning 2 years

~/.atom/git-time-machine/rrev/Time Machine - datum.cjsx ⓘ 0 ⚠ 0 ⓘ 0 1:1 LF ⚠ 1 deprecation UTF-8 CoffeeScript (JSX) git+ 0 files

Figure 2.7: A screenshot of the current version according to the project's website.



fix order of annotation arguments for deprecated tree traversal #6522

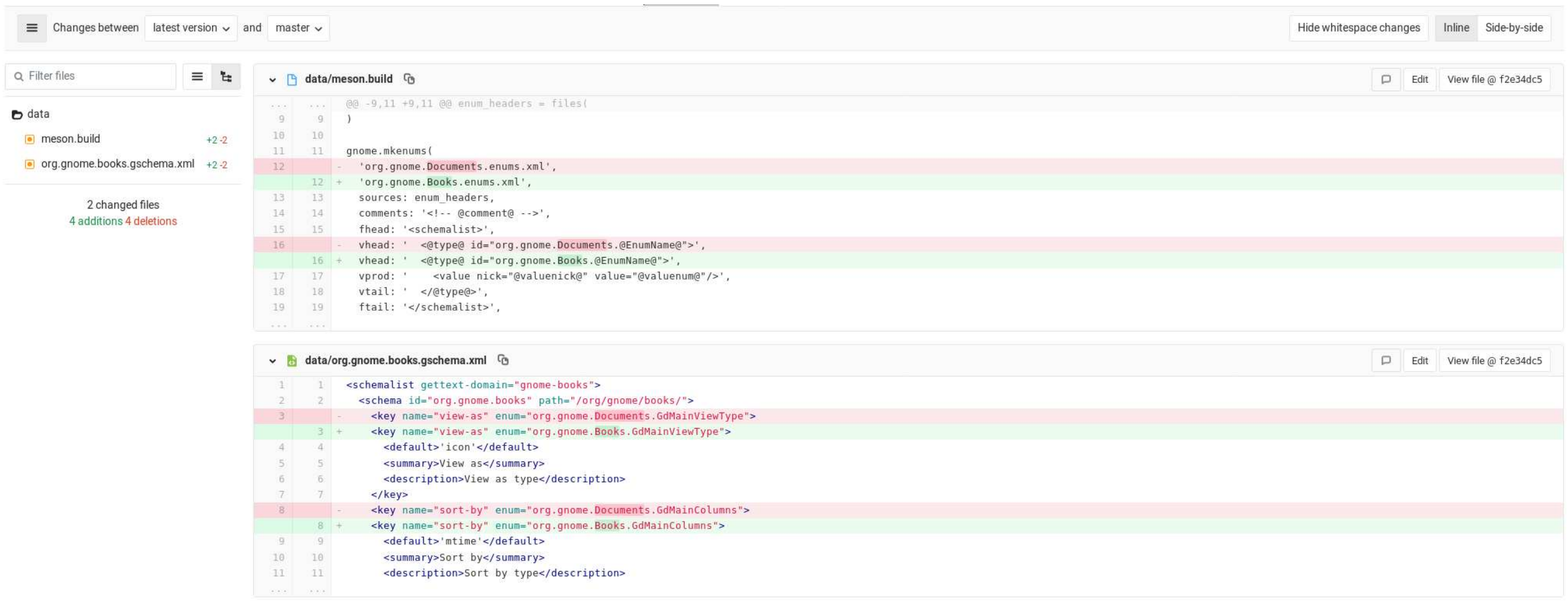
lrytz merged 1 commit into scala:2.13.x from m-sp:2.13.x on Apr 15, 2018

+2 −2

4 src/reflect/scala/reflect/api/Trees.scala

```
@@ -2517,14 +2517,14 @@ trait Trees { self: Universe =>
      *  because pattern matching on abstract types we have here degrades performance.
      *  @group Traversal
      */
-    @deprecated("2.12.3", "Use Tree#traverse instead")
+    @deprecated("Use Tree#traverse instead", "2.12.3")
     protected def itraverse(traverser: Traverser, tree: Tree): Unit = throw new MatchError(tree)

     /** Provides an extension hook for the traversal strategy.
      *  Future-proofs against new node types.
      *  @group Traversal
      */
-    @deprecated("2.12.3", "Use Tree#traverse instead")
+    @deprecated("Use Tree#traverse instead", "2.12.3")
     protected def xtraverse(traverser: Traverser, tree: Tree): Unit = throw new MatchError(tree)

     /** A class that implement a default tree transformation strategy: breadth-first component-wise cloning.
```

Figure 2.8: The same git diff visualized by the GitHub website



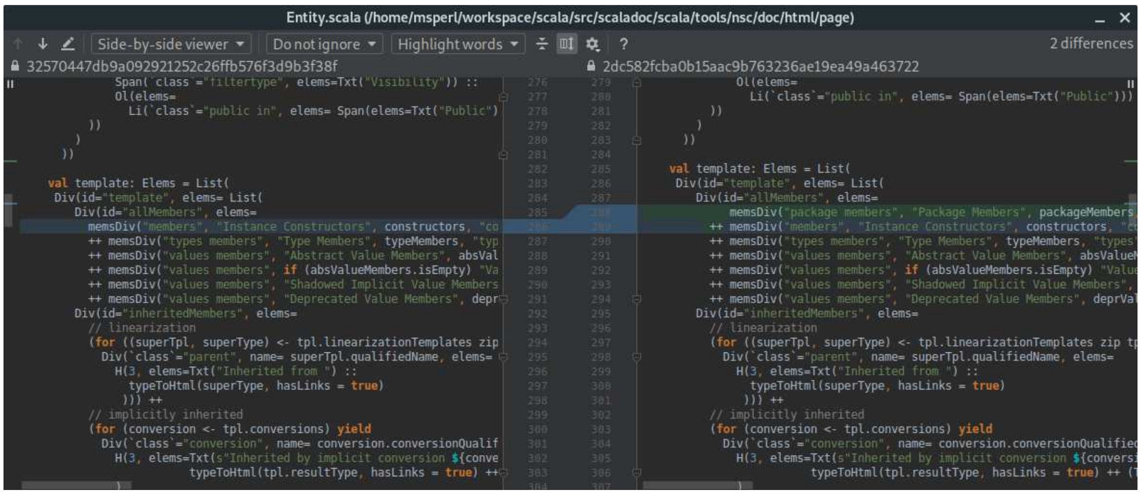
data/meson.build

```
@@ -9,11 +9,11 @@ enum_headers = files(
)

gnome.mkenums(
-  'org.gnome.Documents.enums.xml',
+  'org.gnome.Books.enums.xml',
   sources: enum_headers,
   comments: '<!-- @comment@ -->',
   fhead: '<schemalist>',
-  vhead: '  <@type@ id="org.gnome.Documents.@EnumName@">',
+  vhead: '  <@type@ id="org.gnome.Books.@EnumName@">',
   vprod: '    <value nick="@valuenick@" value="@valuenum@"/>',
   vtail: '  </@type@>',
   ftail: '</schemalist>',
```

data/org.gnome.books.gschema.xml

```
<schemalist gettext-domain="gnome-books">
  <schema id="org.gnome.books" path="/org/gnome/books/">
-    <key name="view-as" enum="org.gnome.Documents.GdMainViewType">
+    <key name="view-as" enum="org.gnome.Books.GdMainViewType">
      <default>'icon'</default>
      <summary>View as</summary>
      <description>View as type</description>
    </key>
-    <key name="sort-by" enum="org.gnome.Documents.GdMainColumns">
+    <key name="sort-by" enum="org.gnome.Books.GdMainColumns">
      <default>'mtime'</default>
      <summary>Sort by</summary>
      <description>Sort by type</description>
```

Figure 2.9: A simple code change visualized by the GitLab UI


Entity.scala (/home/msperl/workspace/scala/src/scaladoc/scala/tools/nsc/doc/html/page)
Side-by-side viewer
Do not ignore
Highlight words
?
2 differences

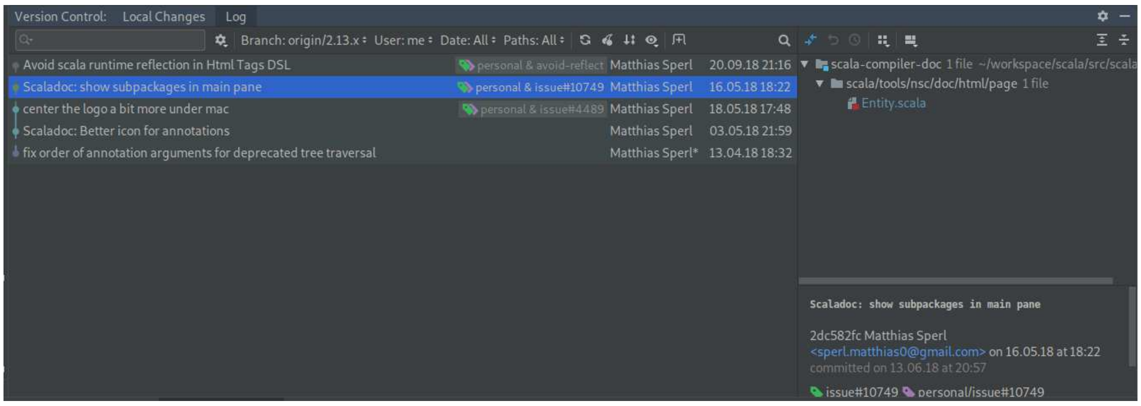
32570447db9a092921252c26ffb576f3d9b3f38f

```
        Span(`class`="filtertype", elems=Txt("Visibility")) ::
        Ol(elems=
          Li(`class`="public in", elems= Span(elems=Txt("Public")
      ))
    )
  ))

val template: Elems = List(
 Div(id="template", elems= List(
   Div(id="allMembers", elems=
     memsDiv("members", "Instance Constructors", constructors, "co
     ++ memsDiv("types members", "Type Members", typeMembers, "typ
     ++ memsDiv("values members", "Abstract Value Members", absVal
     ++ memsDiv("values members", if (absValueMembers.isEmpty) "Va
     ++ memsDiv("values members", "Shadowed Implicit Value Members
     ++ memsDiv("values members", "Deprecated Value Members", depr
   Div(id="inheritedMembers", elems=
     // linearization
     (for ((superTpl, superType) <- tpl.linearizationTemplates zip
       Div(`class`="parent", name= superTpl.qualifiedName, elems=
         H(3, elems=Txt("Inherited from ") ::
           typeToHtml(superType, hasLinks = true)
          ))) ++
     // implicitly inherited
     (for (conversion <- tpl.conversions) yield
       Div(`class`="conversion", name= conversion.conversionQualif
         H(3, elems=Txt(s"Inherited by implicit conversion ${conve
                    typeToHtml(tpl.resultType, hasLinks = true) ++
```

2dc582fcba0b15aac9b763236ae19ea49a463722

```
        Ol(elems=
          Li(`class`="public in", elems= Span(elems=Txt("Public")))
      ))
    )
  ))

val template: Elems = List(
 Div(id="template", elems= List(
   Div(id="allMembers", elems=
     memsDiv("package members", "Package Members", packageMembers,
     ++ memsDiv("members", "Instance Constructors", constructors, "c
     ++ memsDiv("types members", "Type Members", typeMembers, "types
     ++ memsDiv("values members", "Abstract Value Members", absValueM
     ++ memsDiv("values members", if (absValueMembers.isEmpty) "Value
     ++ memsDiv("values members", "Shadowed Implicit Value Members",
     ++ memsDiv("values members", "Deprecated Value Members", deprVal
   Div(id="inheritedMembers", elems=
     // linearization
     (for ((superTpl, superType) <- tpl.linearizationTemplates zip tp
       Div(`class`="parent", name= superTpl.qualifiedName, elems=
         H(3, elems=Txt("Inherited from ") ::
           typeToHtml(superType, hasLinks = true)
          ))) ++
     // implicitly inherited
     (for (conversion <- tpl.conversions) yield
       Div(`class`="conversion", name= conversion.conversionQualifie
         H(3, elems=Txt(s"Inherited by implicit conversion ${conversi
                    typeToHtml(tpl.resultType, hasLinks = true) ++ (T
```

Figure 2.10: A simple code change visualized by the IntelliJ IDEA diff viewer



```
Avoid scala runtime reflection in Html Tags DSL                 personal & avoid-reflect  Matthias Sperl   20.09.18 21:16
Scaladoc: show subpackages in main pane                         personal & issue#10749    Matthias Sperl   16.05.18 18:22
center the logo a bit more under mac                            personal & issue#4489     Matthias Sperl   18.05.18 17:48
Scaladoc: Better icon for annotations                                                     Matthias Sperl   03.05.18 21:59
fix order of annotation arguments for deprecated tree traversal                           Matthias Sperl*  13.04.18 18:32
```

```
▼ scala-compiler-doc 1 file ~/workspace/scala/src/scala
  ▼ scala/tools/nsc/doc/html/page 1 file
      Entity.scala
```

```
Scaladoc: show subpackages in main pane

2dc582fc Matthias Sperl
<sperl.matthias0@gmail.com> on 16.05.18 at 18:22
committed on 13.06.18 at 20:57

issue#10749 personal/issue#10749
```

Figure 2.11: The git history visualized by the IntelliJ IDEA diff viewer

3 Methodology

This thesis is in the realm of design sciences and follows its principals outlined by Von Alan et al. [55]. The underlying method of this science enables researchers to „improve and understand the behavior of aspects of Information Systems“ [55]. Two primary activities are performed to achieve this goal of improving Information Systems.

1. „the creation of new knowledge through design of novel or innovative artifacts (things or processes)“
2. „the analysis of the artifact’s use and/or performance with reflection and abstraction“

The thesis follows these two parts, the implementation phase followed by an evaluation phase. Before starting with the thesis, there was an implicit planning phase done to have an outline and plan what the goals of the thesis and its prototype are.

3.1 Planning Phase

During this phase a thorough survey of the relevant literature was conducted. The state of the art was then used as a basis to design features and algorithms which are useful for the prototype. An important part of this was to find similar software to get a feeling of what is possible and might be useful.

The results of the literature review can be found in the state of the art, but is also highlighted during the different sections of the implementation chapter.

Since the visualization of movement and migrations is not only studied in the context of version control, successfully applied techniques from other fields have been studied. The idea was to find appropriate interactive documents and graphics which have proven useful and try apply them in the context of code evolution.

3.2 Implementation Phase

This section focuses on the methods used to develop the prototype of *PolyCoDif*. The software was developed in an iterative approach based on the Kanban Methodology [32]. Features defined in the planning phase have been continuously improved in small iterations.

To make it possible to integrate PolyCoDif into multiple different editors and IDEs a client/server approach was chosen. The communication between the two components was implemented via a Representational State Transfer (REST) application programming interface (API) based on the work of Fielding [17].

The client which is responsible for displaying the UI and providing the interactive capabilities needs the capabilities of full visualizations. Therefore, the implementation via HTML and SVG has a lot of advantages and again eases the integration with the backend as well as the integration with different editors.

The backend was conceptualized to make it easy to split the software into an indexing and querying component. For the example projects used in the scenario-based expert evaluation it was not necessary to split the software into two actual processes and therefore, for simplicity reasons, a single process was kept.

Additionally, the API was split into two parts, the query part and the continuous tracking part. The tracking is done directly on a file level but could be later extended and implemented based on the LSP project.

Further implementation details of the algorithms can be found in Chapter 4.

3.3 Evaluation Phase

For the assessment of the prototype, a scenario-based expert evaluation with eight participants is conducted. To make it comparable with similar studies the System Usability Scale (SUS) was used.

Experts work through four scenario-based tests. The tests are designed to highlight the worst-case and best-case scenarios for the developed tool, with the idea that the best-case scenario achieves a great test outcome while the results for the best-case scenario are still acceptable.

A viable software project to use as data sources during the test sessions is selected based ideally on the usage of multiple programming languages and the usage of object-oriented and functional programming principles.

A risk here is that very little open-source projects are using multiple programming languages simultaneously and that it might not be possible to use a proprietary project due to license issues.

The results of the evaluation are finally visualized, interpreted and discussed based on a predefined range of valid answers.

4 Implementation

PolyCoDif is developed in the Scala programming language designed by Odersky et al. [43]. The architecture is separated in a client- and server-side component and is explained in more detail in Section 4.1. After the overview over the system, Section 4.3 explains the design process and gives a selection of proposed visualizations and the reason why they were discarded. Section 4.4 explains the implementation of the unified AST which is the basis for the tree-based difference algorithm (Section 4.5). Even in a tree structure there are large bodies of text which need to be compared, for example comments. Therefore, an implementation of a text-based diff algorithm is needed and explained in detail in Section 4.6. Section 4.7 combines the classic tree-based diff algorithm with continuously gathered information based on category theory [39].

4.1 Architecture

The two parts of the system – client and server – communicate with each other via a REST API [17]. The visualizations are computed on the client-side and can be displayed with a browser but can also be integrated into an IDE, if it supports WebViews. WebViews are Graphical User Interface (GUI) widgets which are used to embed the rendering engine of a browser in native applications. While researching the capabilities of widely used code editors, it became clear that most modern editors supported this technology. Therefore, a HTML based visualization was chosen.

The backend on the other hand has the purpose of doing all the computations, gathering the continuously sent data and combining this data with the data stored in the VCS.

A prototype for storing this continuously gathered data as Git notes [22] is part of the backend but was not part of the evaluation in Chapter 5.

4.1.1 User Interface

The client is written in Scala.js [16]. For visualizations it uses html and `D3.js`[1]. To integrate `D3.js` with Scala some boilerplate code in JavaScript was written.

`D3.js` provides compelling features to build complex data-based visualizations. In the final version of the prototype it was mostly used for the timeline at the bottom of the screen.

In earlier versions of the UI it was also used to describe the *Move operation* based on Sankey diagrams. More details about the implementation of these visualizations can be found in Section 4.3.

To enable fast iterations resulting in continuous improvements of the prototype, both frontend and backend were developed in a common Git repository. As a build tool for this single repository `sbt`[2] was used.

[1] https://d3js.org/
[2] https://www.scala-sbt.org/

4.1.2 REST API

The utility of the REST API was to expose the services provided by the backend to the browser based frontend. As the frontend was querying specific resources like the history, commits and the file tree from the backend, the REST approach was a natural fit.

The basic types managed in a repository can be viewed as a hierarchical set of resources. The topmost level is the project with the respective sub-levels. The project consists of an outline and a history. The outline gives an overview of the files and directories in a given point in time and the history consists of the changes over time in the form of commits. Combining the file axis and the time axis a patch can be queried and be referenced exactly. These endpoints take a path representation and a time range and return the differences of this patch in a JavaScript Object Notation (JSON) format.

Because *PolyCoDif* can produce a text-based and a tree-based edit-script the patch endpoints are split in a semantic and textual endpoint.

4.1.3 Delta Module

This module is responsible for providing the laws defined and discussed in Section 4.7.

To ensure that the code is a correct implementation of these laws a property-based test system named ScalaCheck[3] is used. This test framework is based on the ideas discussed by Claessen and Hughes [14] in this paper about QuickCheck.

Property-based testing randomly generates inputs for function and ensures that the function succeeds on every single input. The *reverse* function for a standard list definition is used as a concrete example to explain the principals of property-based testing. This *reverse* function can be defined by three laws:

First the *reverse* of a list with exactly one element, is the same list.

$$\forall [x].reverse([x]) = [x] \tag{4.1}$$

The second law says that concatenating two arbitrary lists xs and ys and reversing the result is the same as reversing each list individually and concatenating the lists afterwards.

$$\forall xs, ys.reverse(xs + +ys) = (reverse(xs)) + +(reverse(ys)) \tag{4.2}$$

Finally, the third law states that reversing a list twice yields exactly the initial lists.

$$\forall xs.reverse(reverse(xs)) = xs \tag{4.3}$$

As one can see all these laws begin with a universal quantification clause. So to exhaustively test the reverse function, it is either necessary to formally proof the function (in this case by induction) or to input all values from the domain of the function. A simpler solution than exhaustively listing the domain is to generate an arbitrary, but limited, number of values and assume that if it works for some arbitrary number of values, it will work for all values.

Of course this is no formal proof but is sufficient for most industrial strength programs and goes far beyond what's possible with standard unit testing [5, 14].

[3] http://www.scalacheck.org/

4.1.4 Git Based Backend

At its core Git is a very well performing key-value store. It is useful to store arbitrary blobs and Git will create a unique key for each blob. Based on this simple object store Git layers higher-level concepts.

File Object File Objects are simple data containers which directly represent the content of files.

Tree Object A tree is used to store a group of files together and associate them with metadata. To get the content of each individual file a pointer to a another file object is part of the tree object.

Commit Object What is commonly known as a commit is just another object in the same key-value store. The object associates tree objects with commit level metadata. Examples of such metadata are the author, the date or the commit message. In addition a commit can reference a previous commit.

From these basic objects one can see that a Git repository is a complex graph store mapped on a simple key-value data model. Git freely exposes these low-level objects and can therefore be extended to persist additional metadata quite naturally. [11]

One built-in exploitation of this system are Git notes. The idea of notes is to attach additional meaning and information to commits. The idea of PolyCoDif is to attach the continuously gathered change to commits based on notes. Since Git does not prescribe any format a JSON string was used. While easy to implement this had the additional advantage to simplify debugging of the storage layer.

Since Git notes can be shared with other repositories with the same basic *push* and *pull* operations, this approach enables the seamless sharing of key-press level or save-file level change events with other software engineers without further modifications to both git and PolyCoDif [22].

4.1.5 Analysis API

This is the core module containing the Change Distiller algorithm by Fluri et al. [19] and the textual diff algorithm by Myers [40]. The algorithms are explained in depth in Section 4.5 and 4.6.

The Change Distiller implementation was forked from the reference implementation provided by the author and adapted to work with PolyCoDif. Similarly, the text-based algorithm was ported to Scala and is based on the original paper.

4.1.6 Language Analyzer

PolyCoDif provides a dedicated module for each supported programming language to analyze the files containing the source code. These modules contain a lexer, which converts a stream of characters into tokens, and a parser, which converts these tokens to an AST. These are the same steps performed by most compilers as described by Wirth et al. [60] For the conversion from strings to the language specific ASTs open-source libraries can be used. As a final step this is transformed to the unified AST described previously. These modules are explained in brief in the following sections.

Java Analyzer

For simplicity reasons the Java analyzer is based on the open-source *JavaParser* library[4]. This library generates a Java specific parsing tree which is furthermore transformed in the full-fledged unified AST described in Section 4.4.

This transformation is done in a recursive style. For a correct handling with the continuous tracking approach it is of utmost importance to correctly assign the source text ranges from which the individual nodes where generated.

Scala Analyzer

The Scala analyzer was implemented based on the same principles as the Java analyzer. Instead of the *JavaParser* project the text analyzer was implemented based on the *scala.meta* library[5]. The conversion from the Scala AST to the unified AST described in Section 4.4 was again analogous to the Java conversion.

4.2 Semantic Operations

The simple text-based diff algorithm generates a list with two kinds of change operations. The **Add** operation and the **Delete** operation. Besides the `item` that is changed the operation contains the index at which position the element is inserted or deleted. The definition implemented in Scala for this is given below.

```scala
sealed trait TextualDelta
case class Added(index: Int, item: Elem)   extends TextualDelta
case class Removed(index: Int, item: Elem) extends TextualDelta
```

For optimization reasons most algorithms have a third operation which combines the add and delete.

```scala
case class Changed(index: Int, added: Elem, removed: Elem) extends
↪ TextualDelta
```

These same basic operations can be extended to work on trees in a semantic context:

```scala
sealed trait SemanticDelta

case class SemInsert(pos: Position, node: SemanticTree) extends
↪ SemanticDelta
case class SemDelete(pos: Position, node: SemanticTree) extends
↪ SemanticDelta
case class SemMove(oldPosition: Position, newPosition: Position,
↪ node: SemanticTree) extends SemanticDelta

```

[4] https://javaparser.org/
[5] https://scalameta.org/

```
case class SemUpdate(oldPosition: Position,
                     newPosition: Position,
                     delta: Vector[TextualDelta],
                     oldNode: SemanticTree,
                     newNode: SemanticTree) extends SemanticDelta
```

While the `SemInsert` and the `SemDelete` are a straightforward translation form the simple text-based diff, the `SemMove` and the `SemUpdate` are more complex and need some more explanations.

Semantic Move This operation is used when a node is removed from one part of the tree and the exact same node is attached at a different position in the tree. The move data structure consists of three fields. The old position in the tree, the new position in the tree and the node which is moved. While the node is not strictly necessary it is useful when implementing the continuous refinement described in Section 4.7.

Semantic Update The update operation extends the previously described semantic move and is the most generic operation. Just like the move operations it consists of an old and a new position. In addition to that it describes a textual change in the moved node. An example for this kind of operation would be if a method is moved to another class and a new argument is added to the method.

Theoretically this operation is good enough to describe all three previous operations. E.g. a move is just an update with an empty textual delta and an insert is a move from the empty position with only textual additions in the delta. However, the update operation is the most difficult to visualize and understand so there is always a tradeoff when the algorithm should distill a change to an update operation.

These operations are based on the work by Fluri et al. [19]. Since each operation only represents a single change, the concept of a patch-set is again needed to combine individual changes to a complete edit-script between to versions of source code. With that in mind PolyCoDif defines a semantic patch-set as a vector of semantic diff operations.

```
  case class SemanticPatchSet(deltas: Vector[SemanticDelta])
```

The next section deals with the visualization of these semantic patch-sets. Followed by an explanation how to construct such a collection of semantic delta operations.

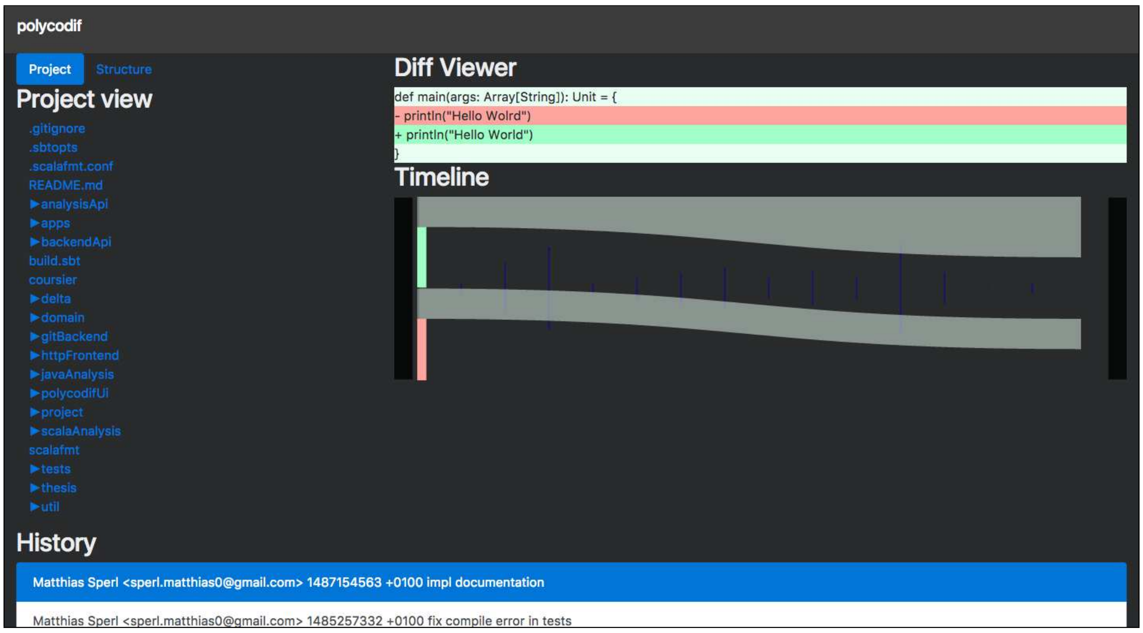
4.3 Visualization

The initial design was inspired by the visualization of population movement diagrams, the analysis of voting transfer and Sankey diagrams. An example of this can be seen in Figure 4.1.

While this worked for small amounts of changes, this became very unreadable and confusing on more realistic examples. On the one hand it proofed troublesome if there were more than a handful of moves. Especially because inevitably moves overlapped and formed a sort of **X**.

Adding insert and remove operations was quite natural in the visualization. An add was represented as green lines on the right-hand side of the Sankey diagram with either no start population on the left hand side or a starting population of zero. The deletion was represented as the reverse.

It was still tractable if the changes were restricted to the aforementioned operations only, but it become especially incomprehensible as soon as the most generic update operation was added.



.gitignore
.sbtopts
.scalafmt.conf
README.md
▶analysisApi
▶apps
▶backendApi
build.sbt
coursier
▶delta
▶domain
▶gitBackend
▶httpFrontend
▶javaAnalysis
▶polycodifUI
▶project
▶scalaAnalysis
scalafmt
▶tests
▶thesis
▶util

```
def main(args: Array[String]): Unit = {
- println("Hello Wolrd")
+ println("Hello World")
}
```

Matthias Sperl <sperl.matthias0@gmail.com> 1487154563 +0100 impl documentation

Matthias Sperl <sperl.matthias0@gmail.com> 1485257332 +0100 fix compile error in tests

Figure 4.1: An early prototype based on Sankey diagrams

The update operation is a combination of move and a text-based diff. So while the move part can be represented similar to the complete move operation, the integration of the classic text diff was too troublesome. In the end, the mismatch between the very text-based process of writing code and the purely visual representation of Sankey diagrams was too big to overcome and unintuitive for users.

The visualization was then replaced with a an evolutionary approach to the existing tools seen in the state of the art section. Instead of focusing on the individual files the main view deals with the individual operations performed on the source code. The restriction to certain files can be performed with the hierarchy pane on the left hand side. An example of this can be found in Figure 4.2.

In the following sections the visualization of each individual operation is described in more detail.

4.3.1 Insert

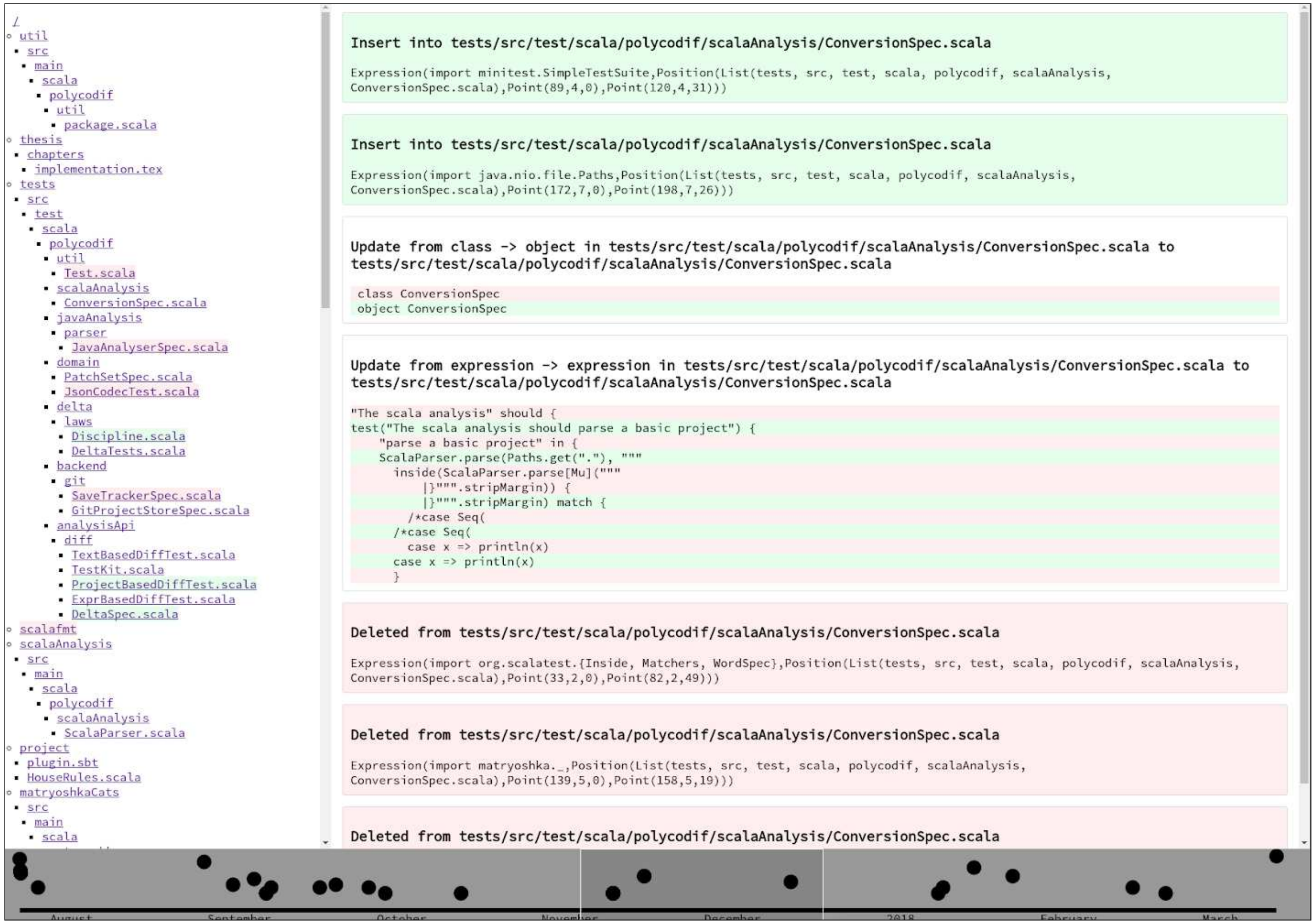
The insert archetype has a green background to make it familiar for users of tools like GitHub or GitLab. The top headline gives some context on which compilation unit the operation is performed, while the body describes the sub tree which is inserted in the file. This can be seen in figure 4.3. Additionally, the position within the source file is indicated to the user.

4.3.2 Delete

In contrast to the insert operation, the delete operation has a red background. The rest follows the same structure as described above.

4.3.3 Move

The header of the move operation is the most important part. It designates which kind of node is moved and from which position to which position. In addition, the body contains the textual representation of the node with a white background to make it clear for users what was moved.



```
/
◦ util
  ▪ src
    ▪ main
      ▪ scala
        ▪ polycodif
          ▪ util
            ▪ package.scala
◦ thesis
  ▪ chapters
    ▪ implementation.tex
◦ tests
  ▪ src
    ▪ test
      ▪ scala
        ▪ polycodif
          ▪ util
            ▪ Test.scala
          ▪ scalaAnalysis
            ▪ ConversionSpec.scala
          ▪ javaAnalysis
            ▪ parser
              ▪ JavaAnalyserSpec.scala
          ▪ domain
            ▪ PatchSetSpec.scala
            ▪ JsonCodecTest.scala
          ▪ delta
            ▪ laws
              ▪ Discipline.scala
              ▪ DeltaTests.scala
          ▪ backend
            ▪ git
              ▪ SaveTrackerSpec.scala
              ▪ GitProjectStoreSpec.scala
          ▪ analysisApi
            ▪ diff
              ▪ TextBasedDiffTest.scala
              ▪ TestKit.scala
              ▪ ProjectBasedDiffTest.scala
              ▪ ExprBasedDiffTest.scala
              ▪ DeltaSpec.scala
◦ scalafmt
◦ scalaAnalysis
  ▪ src
    ▪ main
      ▪ scala
        ▪ polycodif
          ▪ scalaAnalysis
            ▪ ScalaParser.scala
◦ project
  ▪ plugin.sbt
  ▪ HouseRules.scala
◦ matryoshkaCats
  ▪ src
    ▪ main
      ▪ scala
```

Insert into tests/src/test/scala/polycodif/scalaAnalysis/ConversionSpec.scala

```
Expression(import minitest.SimpleTestSuite,Position(List(tests, src, test, scala, polycodif, scalaAnalysis,
ConversionSpec.scala),Point(89,4,0),Point(120,4,31)))
```

Insert into tests/src/test/scala/polycodif/scalaAnalysis/ConversionSpec.scala

```
Expression(import java.nio.file.Paths,Position(List(tests, src, test, scala, polycodif, scalaAnalysis,
ConversionSpec.scala),Point(172,7,0),Point(198,7,26)))
```

Update from class -> object in tests/src/test/scala/polycodif/scalaAnalysis/ConversionSpec.scala to tests/src/test/scala/polycodif/scalaAnalysis/ConversionSpec.scala

```
class ConversionSpec
object ConversionSpec
```

Update from expression -> expression in tests/src/test/scala/polycodif/scalaAnalysis/ConversionSpec.scala to tests/src/test/scala/polycodif/scalaAnalysis/ConversionSpec.scala

```
"The scala analysis" should {
test("The scala analysis should parse a basic project") {
    "parse a basic project" in {
    ScalaParser.parse(Paths.get("."), """
      inside(ScalaParser.parse[Mu]("""
          |}""".stripMargin)) {
          |}""".stripMargin) match {
        /*case Seq(
      /*case Seq(
        case x => println(x)
      case x => println(x)
      }
```

Deleted from tests/src/test/scala/polycodif/scalaAnalysis/ConversionSpec.scala

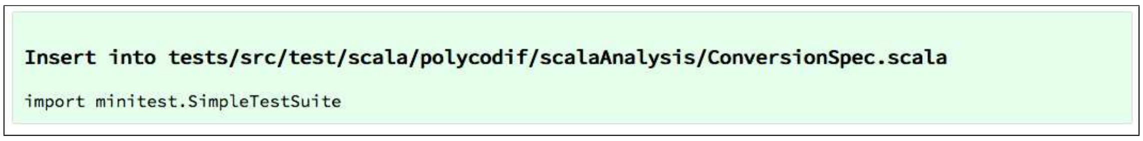
```
Expression(import org.scalatest.{Inside, Matchers, WordSpec},Position(List(tests, src, test, scala, polycodif, scalaAnalysis,
ConversionSpec.scala),Point(33,2,0),Point(82,2,49)))
```

Deleted from tests/src/test/scala/polycodif/scalaAnalysis/ConversionSpec.scala

```
Expression(import matryoshka._,Position(List(tests, src, test, scala, polycodif, scalaAnalysis,
ConversionSpec.scala),Point(139,5,0),Point(158,5,19)))
```

Deleted from tests/src/test/scala/polycodif/scalaAnalysis/ConversionSpec.scala

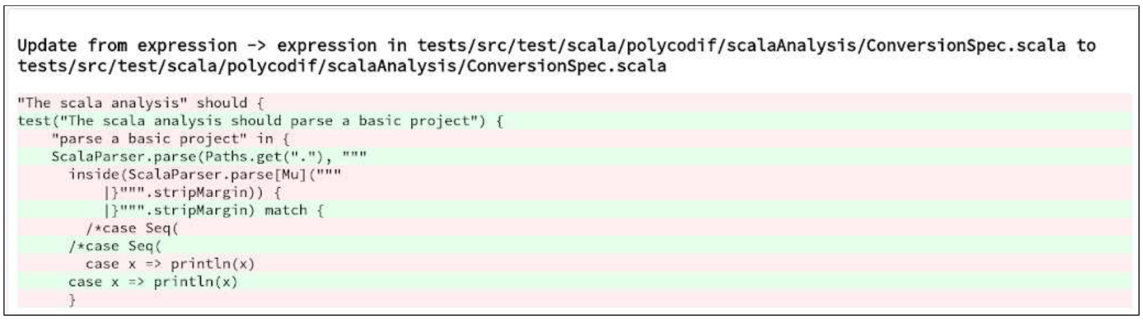
Figure 4.2: A prototype for semantic text-based description of a diff



Insert into tests/src/test/scala/polycodif/scalaAnalysis/ConversionSpec.scala

```
import minitest.SimpleTestSuite
```

Figure 4.3: Prototype visualization of the insert operation



Update from expression -> expression in tests/src/test/scala/polycodif/scalaAnalysis/ConversionSpec.scala to tests/src/test/scala/polycodif/scalaAnalysis/ConversionSpec.scala

```
"The scala analysis" should {
test("The scala analysis should parse a basic project") {
    "parse a basic project" in {
    ScalaParser.parse(Paths.get("."), """
      inside(ScalaParser.parse[Mu]("""
          |}""".stripMargin)) {
          |}""".stripMargin) match {
        /*case Seq(
      /*case Seq(
        case x => println(x)
      case x => println(x)
      }
```

Figure 4.4: Prototype visualization of the update operation

4.3.4 Update

The update operation extends the move operation, while the header is pretty similar the body contains a classical text-based diff view, derived from the textual delta of the operation. An example of this can be seen in figure 4.4. It turned out that this visualization was more familiar and therefore easier to use for software developers which are used to the UIs of GitHub, IntelliJ IDEA and so on. Furthermore, this sort of update visualization easily scales to more than ten operations, whereas the Sankey chart struggled with upwards of five operations.

4.4 Unified Abstract Syntax Tree

To ensure all the algorithms work on as many programming languages as possible PolyCoDif requires a single, shared tree structure as hinted in Section 4.1.6. The focus of this AST is to mostly capture the overall structure of code while leaving individual expressions intact.

The reason for this is, that initial experiments with a very fine-grained level AST have shown that the diff algorithms find moves on arbitrary changes.

For example, given an initial code block of:

```java
public void calculateEditScript(Node<A> left, Node<A> right) {
  fMatch = new HashSet<>();
  TreeMatcher<A> dnm = MatchingFactory.getMatcher(fMatch);
  dnm.match(left, right);
  fLeftToRightMatch = new HashMap<>();
  fRightToLeftMatch = new HashMap<>();
  for (NodePair<A> p : fMatch) {
    fLeftToRightMatch.put(p.getLeft(), p.getRight());
    fRightToLeftMatch.put(p.getRight(), p.getLeft());
  }
  editScript(left, right);
}
```

and afterwards changing it to:

```
public void calculateEditScript(Node<A> left, Node<A> right) {
  fMatch = new HashSet<>();
  TreeMatcher<A> dnm = MatchingFactory.getMatcher(fMatch);
  dnm.match(left, right);
  editScript(left, right);
}

private void initializeHashMaps() {
  otherHashMap = new HashMap<>();
}
```

the algorithm recognizes a move from the construction of the `HashMap`. While theoretically this is correct in practice this is not very useful.

With this in mind, it is quite clear that expressions are only captured up until a level which is equal to either a complete line or a complete block, while definitions are captured on a very detailed level. This in turn helps to recognize a lot of the refactoring operations available in IDEs today. Most either introduce or change definitions like methods, values, or variables. The tree structure based on this information, which was implemented for PolyCoDif, is given in the code Listing 43.

While this captures the main tree, some utility definitions are needed. E.g. since a lot of refactoring operations change specific types, PolyCoDif explicitly models the possible nodes rather strictly.

```
sealed trait Type {
  def bounds: List[Bound]
}
case class SimpleType(name: Identifier, annotations: List[Annotation
↪ ], bounds: List[Bound]) extends Type
case class GenericType(name: Identifier, annotations: List[
↪ Annotation], parameters: Seq[Type], bounds: List[Bound]) extends
↪ Type

sealed trait Bound
case class UpperBound(tpe: Type) extends Bound
case class LowerBound(tpe: Type) extends Bound
```

Additionally, one feature which is quite important in e.g. Java is the annotation support. Lots of software which utilize the spring framework are quite dependent on annotations. Therefore, they warrant their own special treatment in the AST.

```
sealed trait Annotation
case object Public                       extends Annotation
case object Protected                    extends Annotation
case object Private                      extends Annotation
case class TypeAnnotation(tpe: Type)     extends Annotation
case class CodeAnnotation(name: String)  extends Annotation
case class Custom(name: Identifier)      extends Annotation
case object Static                       extends Annotation
case object NoAnnotation                 extends Annotation
```

```
sealed trait SemanticTree {
  def label: String
  def value: String
  def body: List[SemanticTree]
  def annotations: List[Annotation]
  def position: Position
}
sealed trait Definition extends SemanticTree
case class DefDef(name: Identifier, body: List[SemanticTree],
↪ position: Position) extends Definition {
  def label: String = "def"
}
case class ValDef(name: Identifier, body: List[SemanticTree],
↪ position: Position) extends Definition {
  def label: String = "val"
}
case class VarDef(name: Identifier, body: List[SemanticTree],
↪ position: Position) extends Definition {
  def label: String = "var"
}
case class TypeDef(name: Identifier, position: Position) extends
↪ Definition {
  def label: String = "type"
  def body          = Nil
}
case class ClassDef(name: Identifier, body: List[SemanticTree],
↪ position: Position) extends Definition {
  def label: String = "class"
}
case class TraitDef(name: Identifier, body: List[SemanticTree],
↪ position: Position) extends Definition {
  def label = "trait"
}
case class ObjectDef(name: Identifier, body: List[SemanticTree],
↪ position: Position) extends Definition {
  def label: String = "object"
}
case class PackageDef(name: Identifier, body: List[SemanticTree],
↪ position: Position) extends Definition {
  def label: String = "def"
  def annotations   = Nil
}
sealed trait Expr extends SemanticTree {
  def body                          = Nil
  def annotations: List[Annotation] = Nil
}
case class Expression(expr: String, position: Position) extends
↪ Expr {
  def label: String = "expression"
  def value: String = expr
}
```

Listing 4.1: The structure of the unified AST used by PolyCoDif

While this AST should be sufficient for most object-oriented programming languages and for a lot of functional programming languages, one might have issues to model more declarative languages like SQL or prolog. There is always the possibility to extend the AST definition with *custom*, language specific, extensions.

4.5 Tree-based Diff

In this section the tree-based diff algorithm for *PolyCoDif* is described. First the description of the original Change Distiller algorithm by Fluri et al. [19] is given followed by the adaptations done in order to be applicable in a multi-language setting.

This section is using the change operations described above, since they are directly based on the operations used in the original algorithm.

4.5.1 Change Distiller

The algorithm itself is based on the work by Chawathe et al. [12, 13] and describes how one can *distill* change operations from two different versions of source code.

Change distiller operates in two phases:

1. Calculating a „matching“ between the nodes of the old and the new tree
2. Finding a tree edit-script based on the computed matching

The hard part is calculating the best „matchings“ between the nodes. This is done with the algorithm described in the Listing 4.1.

```
Input: trees $T_1, T_2$
Result: final matching set: $M_{final}$
1  initialization;
2  $M_{final} \leftarrow \emptyset$, $M_{tmp} \leftarrow \emptyset$ Mark all nodes in $T_1$ and $T_2$
     unmatched
3  for leaf $x \in T_1$ and leaf $y \in T_2$ do
4      if $match_1(x, y)$ then
5          $M_{tmp} \leftarrow M_{tmp} \cup (x, y, sim_{2g}(v(x), v(y)))$
6      end
7  end
8  Sort $M_{tmp}$ into descending order, according to the leaf-pair
     similarity
9  for leaf-pair similarity $(x, y, sim_{2g}(v(x), v(y))) \in M_{tmp}$
     do
10     $M_{final} \cup (x, y)$
11     Remove all leaf-pairs from $M_{tmp}$ that contain $x$ or $y$
12     Mark $x$ and $y$ matched
13 end
14 Proceed post-order on trees $T_1$ and $T_2$:
15 for unmatched node $x \in T_1$, if there is an unmatched node
     $y \in T_2$ do
16     if $match_2(x, y)$(including dynamic threshold and inner
         node similarity weighting) then
17         $M_{final} \cup (x, y)$
18         Mark $x$ and $y$ matched
19     end
20 end
```

Algorithm 4.1: The Matching part of the Change Distiller algorithm according to Fluri et al. [19].

The `match` part of the algorithm decides how minimal the generated edit-script will be. To ideally accommodate the problems discovered during testing with the original algorithm and a multi-language AST the match procedure has been improved. These improvements will be described in Section 4.5.2. [19]

Based on the „matching“ list the individual operations are computed. The algorithm was initially described by Chawathe et al. [13] and has been adopted verbatim for Change Distiller. A description can be found in Listing 4.2.

```
$E \leftarrow \emptyset, M' \leftarrow M$
for Visit the nodes of $T_2$ in breadth-first order do
    /* this traversal combines the update, insert, align, and
     move phases */
    Let $x$ be the current node in the breadth-first search of $T_2$
    Let $x = p(x)$
    Let $z$ be the partner of $x$ in $M'$
    if $x$ has no partner in $M'$ then
        $k \leftarrow FindPos(x)$
        Append Insert operation to $E$
        Add $(w, x)$ to $M'$ and apply the Insert operation to $T_1$
    else if $x$ is not a root then
        Let $w$ be the partner of $x$ in $M'$
        Let $v = p(w)$ in $T_1$
        if $v(w) \neq v(x)$ then
            Append Update operation to $E$
            Apply Update operation to $T_1$
        end
        if $(y, v) \notin M'$ then
            Let $z$ be the partner of $y$ in $M'$
            $k \leftarrow FindPos(x)$
            Append Move operation to $E$
            Apply Move operation to $T_1$
        end
    $AlignChildren(w, x)$
end
for Do a post-order traversal of $T_1$ /* this is the delete phase */ do
    Let $w$ be the current node in the post-order traversal of $T_1$
    if $w$ has no partner in $M'$ then
        Append Delete operation to $E$
        Apply Delete operation to $T_1$
    end
end
Result: $E$ is a minimum cost edit-script
$M'$ is a total matching
$T_1$ is isomorphic to $T_2$
```

Algorithm 4.2: The algorithm to convert the matching to an edit-script described by Chawathe et al. [13]

4.5.2 Adaption for Multi-language Capabilities

Change Distiller is based on the concept of labels and values. This means that the tree structure is very simple.

```
case class Node(String label, String value, children: List[Node])
```

The translation of the unified AST to this simple node structure is very straight forward but is lacking information e.g. the position. *PolyCoDif* extends this node structure to accommodate additional metadata and the parsed, semantic representation of the source code. The exact definition ca be seen in the following code snippet:

```
case class Node[A](String label, String value, children: List[Node[A
↪ ]], a:A)
```

Change Distiller is limited to only compare nodes with the exact same labels. On the one hand this has the advantage of minimizing the number of inferred update operations, which is desirable since these operations are the most complex for the end user to understand. On the other hand, this leads to unclear situations when e.g. a `class` is converted into a singleton `object`.

As an example, the two versions of source code

```
class Foobar {
  def someMethod:Int = 0
}
```

```
object Foobar {
  def someMethod:Int = 0
}
```

yielded the following three operations with the original algorithm.

1. Insert *object Foobar*
2. Move *someMethod* from *class Foobar* to *object Foobar*
3. Delete *class Foobar*

With the refined comparison algorithms this can be simplified to a single update operation. Label changes are treated with less priority than value changes to still minimize the number of update operations.

4.6 Text-based Diff

Due to the decision that the individual expressions represent a coarse-grained level of change, a text-based diff algorithm is needed to display the difference between two expression nodes.

This kind of diff-operation is integrated as part of the *Update* change operation and is based on the classical Myers-Diff algorithm [40].

The result of this algorithm is an instance of the class *PatchSet*, the definition of which can be found in Listing 4.2.

A detailed summary of the algorithm can be found in Listing 4.3.

```scala
case class PatchSet(deltas: Vector[TextualDelta])

sealed trait TextualDelta {
  def text: String
  def position: Point
}
object TextualDelta {

  case class Added(position: Point, text: String) extends
  ↪ TextualDelta {
    assert(text.nonEmpty, "Added␣may␣not␣be␣empty")
  }
  case class Removed(position: Point, text: String) extends
  ↪ TextualDelta {
    assert(text.nonEmpty, "Removed␣may␣not␣be␣empty")
  }
}
```

Listing 4.2: A simplified definition of a text only patchset.

```
Data: Constant $MAX \in [0, M + N]$
Var V: $Array[-MAX..MAX]$ of Integer
1  for $D \leftarrow 0$ to $Max$ do
2      for $k \leftarrow -D$ to $D$ in steps of 2 do
3          if $k = -D \vee k \neq D \wedge V[k-1] < V[k+1]$ then
4              $x \leftarrow V[k+1]$
5          else
6              $x \leftarrow V[k-1] + 1$
7          end
8          $y \leftarrow x - k$
9          while $x < N \wedge y < M \wedge a_{x+1} = b_{y+1}$ do
10             $(x, y) \leftarrow (x+1, y+1)$
11         end
12         $V[k] \leftarrow x$
13         if $x \geq N \wedge y \geq M$ then
14             Length of a shortest edit-script is D
15             Stop
16         end
17     end
18 end
19 Length of a shortest edit-script is greater than MAX
```

Algorithm 4.3: Description of the Myers algorithm given in [40].

The original paper proposes several improvements over this naive implementation, and in fact PolyCoDif uses a library which implements a more space-efficient version of the algorithm. Nevertheless to understand the concepts behind it, this version seems good enough.

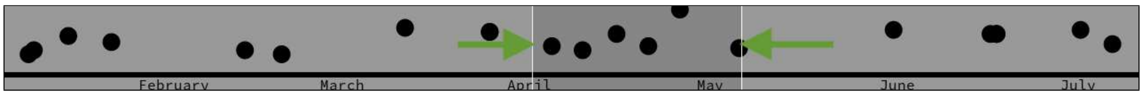


Figure 4.5: Timeline with explicit markings for the drag and drop functionality

4.7 Continuous Refinement of the Patch-set

Calculating the optimal edit-scripts is computationally very expensive. While the algorithms for text-based edit distances are in the range of linear operations, the algorithms for tree-based edit distances are at least in $\mathcal{O}(n^3)$. Where n is the number of nodes in the tree.

Therefore, the more similar the new project state is to the original state, the faster it is to compute a meaningful patch-set. Nevertheless, to be of practical usefulness PolyCoDif needs to be able to deal with longer time ranges.

The continuously captured events are the smallest time ranges captured in the application and PolyCoDif builds upon these to combine the small time ranges to bigger diffs which can be used to compare versions which are apart longer.

To efficiently combine this small diffs and display the tree edit-scripts this approach proofed very useful.

In the UI this features is exposed via the timeline. In figure 4.5 the green arrows point to the borders of the current selection. This selection confirms to the history of commits and can be increased and decreased via *drag and drop*.

In the following sections this thesis explores first the laws of mathematical groups, additionally defines laws for patches and finally the implementation of these interfaces for the text-based edit-scripts and for the tree-based edit-scripts.

4.7.1 Patch-set as a Group

For a sound combination of differences, the patch-set needs to satisfy the four requirements for a mathematical group:

- Closure
- Associativity
- Identity
- Inverse element

To formally define the groups requirements, a notion of equality for edit-scripts is required. A naive implementation of the equality might compare each individual element of the edit-script and if all elements are equal the edit-script itself can be deemed equal.

This simple definition of equality is too strict to work in practice. For example, given an edit-script $\{Add(position = 0, "a"), Add(position = 1, "b")\}$ which inserts the character *a* in position 0 and inserts *b* in position 1 and given an edit-script $\{Add(position = 0, "b"), Add(position = 0, "a")\}$, the simple algorithm for equality would fail. Nevertheless, if both edit-scripts are applied to an empty document the result would in both cases be `"ab"`.

From this example a more complete definition of equality can be derived.

Two edit-scripts *p* and *p'* are equal if and only if the application of the edit-script on any input elements yields the same output.

So given a function apply(p, s) which applies a patch-set *p* to the element *s* the following is valid:

$$p = p' \iff \forall s.(apply(p, s) == apply(p', s)) \tag{4.4}$$

In short this means two patch-sets are equal if and only if both patch-sets applied to any arbitrary value yield the same result.

With this definition of equality, the definition of the group laws can be attempted.

Closure

$$p_1 \cdot p_2 = p \tag{4.5}$$

The closure operation helps define the range and domain of our operation. In practice this means that the domain of our operation is well defined and that it is impossible to get „stuck“.

Associativity

$$(a \cdot b) \cdot c = a \cdot (b \cdot c) \tag{4.6}$$

Associativity is required to make sure that it is irrelevant in which order the individual patch-sets are combined as long as the overall order is not violated.

Identity

$$a \cdot e = e \cdot a = a \tag{4.7}$$

The identity element is the element which does not change the outcome. For example, in the group of all integer numbers combined with the *addition* operator the identity element is 0.

Inverse Element

$$a \cdot a^{-1} = a^{-1} \cdot a = e \tag{4.8}$$

The inverse element is useful in the case of PolyCoDif because of the way the timeline is visualized. A user might add additional elements to the patch-set or subtract elements from it which in other systems needs to be handled by completely recalculating the patch-set. In the case of PolyCoDif the algorithm can combine the currently displayed patch-set with additional changes. This is used when the user extends the selection in the timeline. When the user shrinks the selection, the algorithm can then use the inverse element and append it again to the current patch-set.

4.7.2 Delta Laws

In addition to the laws of the mathematical group, two more laws can be defined to precisely describe the operation Δ, which is responsible for calculating the edit-script between two entities. This section defines this function Δ which computes the patch-set for two arbitrary elements and a function Δ^{-1} which applies a patch to an element.

In combination with the equality definition and the group laws we can now define the distributive property of the patch function as follows:

$$\forall abc(\Delta(a, b) \cdot \Delta(b, c)) = \Delta(a, c) \tag{4.9}$$

Therefore, we can compute a patch-set for a large span in the timeline with combining small changes to large changes.

The efficiency of the approach depends very much on the efficiency of the Δ function.

To make a claim about the complexity of this function one needs to keep in mind that the total complexity of calculating an edit-script for a small amount of nodes and afterwards combining this with a number of fine granular changes can never be lower than calculating the total edit-script for all changes.

If it would be smaller this thesis would have found a more efficient Change Distiller algorithm.

However, even if in practice this is not an overall performance improvement an incremental UI can be implemented based on top of this. Especially the visualization chosen for the history requires the ability to quickly adapt an existing patch.

4.7.3 Textual Patch-set as a Group

Given the previously described laws and the definition of equality the simplest correct implementation of a combination operation for a textual patch-set is the concatenation of the underlying vector. Informally this can be made quite clear since the definition of a patch-set is a sequence of patch operations. For example, given the patch-set A:

$$A = [ins(a,0), ins(b,1), del(a,0)] \tag{4.10}$$

and the patch-set B:

$$B = [ins(a,0), ins(c,1)] \tag{4.11}$$

the combination of the two patch-sets is

$$A + B = [ins(a,0), ins(b,1), del(a,0), ins(a,0), ins(c,1)] \tag{4.12}$$

In practice one can do a lot of optimizations to simplify the resulting patch-set. Let's take the above concatenation as an example. Deleting a character $del(a,0)$ and immediately afterwards inserting the same character at the same position $ins(a,0)$ can be simplified to an empty patch-set $\emptyset$.

So the new minimal patch-set would be:

$$A + B = [ins(a,0), ins(b,1), ins(c,1)] \tag{4.13}$$

The more such optimizations are implemented the more expensive the calculation of the concatenation is. However, the more operations are in the final patch-set the more complex the final visualization is. Therefore, the tradeoff needs to be chosen carefully. For the prototype only very simple optimizations where implemented. A more thorough and sound basis for this can be found in the works of Mimram and Di Giusto [39].

4.7.4 Semantic Patch-set as a Group

Analog to the lawful combination of textual patch-sets a combination of two semantic patch-sets can be implemented. Again, the simplest implementation for this operation is the concatenation of the underlying vector.

Like above there are several optimizations possible to minimize the amount of operations in the resulting patch-set. Since these optimizations are more crucial to the prototype and are not as well studied in the literature, the following parts will describe them in more details.

Insert-Delete Optimization Just like in the textual case an insert at position x followed by a deletion at position x can be optimized to the empty patch-set $\emptyset$.

Delete-Insert Optimization The opposite direction can also trivially be optimized to the empty patch-set.

Move-Delete Optimization A move from position x to position y followed by a deletion of the node at position y can be optimized to the immediate deletion of the node at position x.

Insert-Move Optimization An insert operation at position x followed by a move from x to position y can be replaced with an insert at y.

Move-Move Optimization A move from x to y followed by a move form y to z can be replaced with a direct move from x to z.

Update-Delete Optimization An update from position x to position y followed by a deletion of the node at position y can be optimized to the immediate deletion of the node at position x.

Insert-Move Optimization An insert operation at position x followed by an update from x to position y with the textual delta d can be replaced with an insert at y where the delta d is applied to the insert node.

Update-Move Optimization A update from x to y followed by a move form y to z can be replaced with a direct update from x to z.

Move-Update Optimization A move from x to y followed by an update form y to z can be replaced with a direct update from x to z.

Update-Update Optimization An update $upd(from, x, d_1)$ combined with an update $upd(x, to, d_2)$ can be optimized to an update $upd(from, to, d_1 \oplus d_2)$ where the $\oplus$ is the textual concatenation discussed in the previous section.

4.7.5 Capturing of Continuous Change Events

Research in the realms of continuously capturing changes like the one done by Hattori and Lanza [25] can be used to implement a production ready IDE plugin.

For the purposes of this thesis a very simplified version of this was implemented. IntelliJ IDEA was used as a platform. A plugin can register with the IDE so that it gets informed on each individual key press. When this event is triggered the plugin queries the current position of the cursor.

The combination of the key and the position information is converted to a JSON format and sent to the PolyCoDif server via an HTTP connection.

Since the focus of the thesis was the combination of these continuous changes with a tree-based approach the plugin captures only the most important type of events, namely *inserts*. One notable omission is the capturing of *delete* events. In future research the module can be extended to capture more events.

In addition the granularity of changes is a problem which warrants further research. It might not be necessary to capture each individual key stroke. A better approach might be to capture batches of changes so that the backend system is not overloaded and can spend more processing power on the combination of patch-sets.

Implementation Based on the Language Server Protocol

One other notable improvement for the future might be an implementation of the protocol based on the language server protocol published by [35]. The language server implements a *DidChange-TextDocument* notification which has all the information which is required by PolyCoDif. This can be seen by the definition of the interface given in the Listing 37. Currently there is no LSP implementation for IntelliJ IDEA but it would immediately allow the integration of PolyCoDif in multiple editors.

```
interface DidChangeTextDocumentParams {
  /**
   * The document that did change. The version number points
   * to the version after all provided content changes have
   * been applied.
   */
  textDocument: VersionedTextDocumentIdentifier;

  /**
   * The actual content changes. The content changes describe single
   ↪  state changes
   * to the document. So if there are two content changes c1 and c2
   ↪ for a document
   * in state S10 then c1 move the document to S11 and c2 to S12.
   */
  contentChanges: TextDocumentContentChangeEvent[];
}

/**
 * An event describing a change to a text document. If range and
 ↪ rangeLength are omitted
 * the new text is considered to be the full content of the document
 ↪ .
 */
interface TextDocumentContentChangeEvent {
  /**
   * The range of the document that changed.
   */
  range?: Range;

  /**
   * The length of the range that got replaced.
   */
  rangeLength?: number;

  /**
   * The new text of the range/document.
   */
  text: string;
}
```

Listing 4.3: The LSP definition of a change event.

5 Evaluation

This chapter contains the evaluation of the developed prototype and whether it answers the research questions from 1.2. For this reason, a group of experts are questioned to rank the tool on a scale based on the SUS. In the first part of this chapter the test plan and the proposed scenarios are described in detail, followed by the presentation of the results gathered during the execution of the scenario-based expert evaluation.

5.1 Scenario-based Expert Evaluation

The scenario-based expert evaluation can be separated into two logical sections. While the first gathers data about the demographic of the users, the second evaluates if PolyCoDif is useful for the proposed scenarios. Before going into detail about the scenario-based expert evaluation, first the goal and the scope will be clearly defined. In the following sections the questions about the metadata and the questions about the comparison with other tools are described. The complete questionnaire is attached in Appendix A.2.

The test plan serves as the basis for the expert evaluation and the test is performed based on a prototype of the developed tool. The scenarios where performed with the unreleased *PolyCoDif* to evaluate the visualizations, speed and general usability.

5.1.1 Goal

Due to the first research question being answered in the implementation section, the scenario-based expert evaluation will focus on the second question.

- How do developers benefit from a difference viewer based on continuous AST change detection?

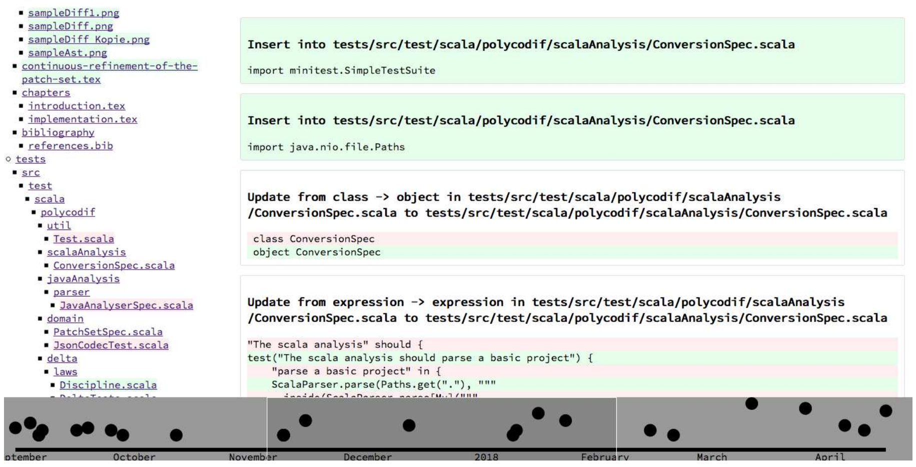
5.1.2 Scope

The scenario-based expert evaluation focuses on the usability of the presentation of the gathered data. For that reason, using the tool during development to gather continuous change sets is out of scope. A configured and running version of *PolyCoDif* will be provided to the participants of the scenario-based expert evaluation. The integration with an IDE like IntelliJ IDEA is also not the focus of the test, therefore the UI will be displayed in a browser as seen in Figure 5.1.

Each scenario is prepared as a dedicated Git repository. Changing between those repositories is done by the moderator because it would involve configuration and setup which is out of scope. Explicitly in scope are the three main parts of the UI. In most cases the user will need the combination of all three UI components – the tree-based project file viewer on the left, the history of the repository on the bottom and the description of the change in the center of the UI.

5.1.3 Methodology

The participants will work through the tasks in random order. While describing each phase of the test, the participants are listed first. The italics font marks the participant who is actively working in the specific phase.



- sampleDiff1.png
- sampleDiff.png
- sampleDiff Kopie.png
- sampleAst.png
- continuous-refinement-of-the-patch-set.tex
- chapters
  - introduction.tex
  - implementation.tex
- bibliography
  - references.bib
- tests
  - src
    - test
      - scala
        - polycodif
          - util
            - Test.scala
          - scalaAnalysis
            - ConversionSpec.scala
          - javaAnalysis
            - parser
              - JavaAnalyserSpec.scala
          - domain
            - PatchSetSpec.scala
            - JsonCodecTest.scala
          - delta
            - laws
              - Discipline.scala

Insert into tests/src/test/scala/polycodif/scalaAnalysis/ConversionSpec.scala

```
import minitest.SimpleTestSuite
```

Insert into tests/src/test/scala/polycodif/scalaAnalysis/ConversionSpec.scala

```
import java.nio.file.Paths
```

Update from class -> object in tests/src/test/scala/polycodif/scalaAnalysis /ConversionSpec.scala to tests/src/test/scala/polycodif/scalaAnalysis/ConversionSpec.scala

```
class ConversionSpec
object ConversionSpec
```

Update from expression -> expression in tests/src/test/scala/polycodif/scalaAnalysis /ConversionSpec.scala to tests/src/test/scala/polycodif/scalaAnalysis/ConversionSpec.scala

```
"The scala analysis" should {
test("The scala analysis should parse a basic project") {
    "parse a basic project" in {
    ScalaParser.parse(Paths.get("."), """
```

Figure 5.1: Overview of a simple graphical representation of a diff created by PolyCoDif.

Preparations before the tests

Participants: *moderator*

- Make sure the required software is installed on the test machine.
- Checkout the four Git repositories containing the scenarios.
- Start the prototype for each scenario.
- Open four browser tabs, each pointing to one of the scenarios, shuffle the order of the browser tabs.
- Open an additional browser window containing the questionnaire about the metadata.

Introduction to the session

Participants: *moderator*, end-user

The moderator gives an introduction the end-user, explaining the scope of the developed tool.

Answering the metadata questionnaire

Participants: *end-user*, moderator

The end-user is answering questions about their person in respect to general development experience and diff tools as detailed in Section 5.1.4.

Introduction to the tool

Participants: *end-user*, moderator

The end-user is provided a concise manual for *PolyCoDif*. They are not yet allowed to actually use the tool.

Execution of the tasks

Participants: *end-user*, moderator

The end-user is executing the tasks in random order as described in Section 5.1.5.

Debriefing

Participants: *end-user*, moderator

The end-user is asked to answer questions about the tool based on SUS and described in Section 5.1.5. In addition the user can give additional feedback and suggestions how to improve *PolyCoDif*. The result of this open questions will be transcribed and added in the appendix.

5.1.4 Metadata

Before conducting the interview, a number of metadata was gathered about the participant. Mostly this was about the experience in software development:

- Levels of experience
- Preferred tool to review code
- Preferred tool to write code
- Experience with
  - Scala
  - Java
  - JavaScript

In addition, the preferred tools for reviewing and writing code was question. This was done to help future work when comparing PolyCoDif to similar tools.

Furthermore, the age and gender with which the participant was recorded. However, since the number of participants was too small this turned out to not be useful data and is therefore not described in details. More details here can be found in the next section.

5.1.5 Comparison of the Diff Viewer

The scenario-based expert evaluation will focus on two use cases of diff tools. **Code reviews** are used to check if the code fits the overall architecture and code style. Additionally, it is used to spread knowledge through the development team and to ensure that the code is written in an understandable manner.

During development of code it is often necessary to understand why and when a certain line of code was introduced. Multiple software engineers working in the same places in the code lead to a lot of **code churn**. These hot-spots are often times changed multiple times and therefore later commits might obfuscate the original reason.

For both use cases the user will be presented with a best-case and a worst-case scenario.

The participant has to rate each tool based on the SUS developed by Brooke [8]. The SUS requires the participants to rate 10 questions on a scale from Strongly Agree to Strongly disagree. The questions are taken verbatim to make it easier to compare the results of this scenario-based expert evaluation with similar research.

1. “I think that I would like to use this system frequently.”
2. “I found the system unnecessarily complex.”
3. “I thought the system was easy to use.”
4. “I think that I would need the support of a technical person to be able to use this system.”
5. “I found the various functions in this system were well integrated.”
6. “I thought there was too much inconsistency in this system.”
7. “I would imagine that most people would learn to use this system very quickly.”
8. “I found the system very cumbersome to use.”
9. “I felt very confident using the system.”
10. “I needed to learn a lot of things before I could get going with this system.”

Each item contributes between 0 and 4 points to the overall score. While for question 1, 3, 5, 7 and 9 *Strongly agree* is equivalent to 4 points and *Strongly disagree* is equivalent to 0 points, for question 2, 4, 6, 8 and 10 the opposite is true. The sum of each individual score is multiplied by 2.5 so that the final score is in the range between 0 and 100[8].

In addition, the user will be asked to assess if they would use *PolyCoDif* again in the future for similar scenarios. These questions will again be on a scale from *Strongly Agree* to *Strongly Disagree*.

1. I found the system was useful to answer the question for scenario X.
2. I would consider using the system in the future if I encounter a similar scenario.

5.1.6 Scenario Description

In the following section a description for the best-case and worst-case scenarios is given. In the appendix there is a list of Git patches to reproduce the Git repositories for these scenarios. Each scenario will have an implicit commit zero, which adds the project file in their initial state.

For each scenario the user needs to answer one question and each question has a defined set of criteria whether the question is answered correctly or not.

Code Review

The code review scenario always asks the question if a certain set of code changes result in a semantically different code or not. It is split in two parts.

CRB The **best-case scenario** will focus on a refactoring without any semantic code change. For this, the end-user will be presented with a sample project based on the *parseback* project[1] with a number of different commits. As with every scenario commit zero is the import of the project *parseback*.

The first proper commit will rename a class and its usages. In this case the

[1] https://github.com/djspiewak/parseback/tree/8aeb1a95a536dc2a07ae7cda0547a805d4fa6dd0

```
final class ParserId[A](val self: Parser[A])
```

is redefined as

```
final class ParserIdentification[A](val self: Parser[A])
```

The diff of this change is printed in Listing 5.1. It is a change in the file *MemoTable.scala* and in the file *parsers.scala*. In the first file the class *ParserId* is renamed. Each usage of this identifier is subsequently updated to reflect the new name. In the second file the class *ParserId* is used as well. Instead of changing each individual usage, the import is changed to an import with an alias.

Previously *renderNonterminal* was defined as a nested method as a part of *renderCompact*. This commit moves the inner method to the same level as *renderCompact* but adds the *private* keyword. The refactoring is equivalent to the way the scala compiler desugars this feature and is therefore – from a semantic point of view – irrelevant. To reproduce the project in each stage the list of patches is given below. Each step is a single git commit and consists of multiple changes.

Code Review – Scenario 1 The scenario consists of three distinct changes, which are listed in the following steps.

The first commit, described by Listing 5.1, shows the renaming of the class *ParserId*.

```
diff --git a/core/src/main/scala/parseback/MemoTable.scala b/core
↪ /src/main/scala/parseback/MemoTable.scala
index 5ed2bb9..dd2033f 100644
--- a/core/src/main/scala/parseback/MemoTable.scala
+++ b/core/src/main/scala/parseback/MemoTable.scala
@@ -33,10 +33,10 @@ private[parseback] sealed abstract class
↪ MemoTable {

 private[parseback] object MemoTable {

-  final class ParserId[A](val self: Parser[A]) {
+  final class ParserIdentification[A](val self: Parser[A]) {

     override def equals(that: Any) = that match {
-      case that: ParserId[_] => this.self eq that.self
+      case that: ParserIdentification[_] => this.self eq that.
↪ self
       case _ => false
     }

@@ -48,14 +48,14 @@ private[parseback] final class
↪ InitialMemoTable extends MemoTable {
   import MemoTable._

   // still using the single-derivation optimization here
-  private val derivations: HashMap[(MemoTable, ParserId[_]), (
↪ Char, Parser[_])] = new HashMap(16)     // TODO tune capacities
-  private val finishes: HashMap[(MemoTable, ParserId[_]),
↪ Results.Cacheable[_]] = new HashMap(16)
```

```
+   private val derivations: HashMap[(MemoTable, ParserIdentification[_]), (Char, Parser[_])] = new HashMap(16)     // TODO tune capacities
+   private val finishes: HashMap[(MemoTable, ParserIdentification[_]), Results.Cacheable[_]] = new HashMap(16)

    def derived[A](from: Parser[A], c: Char, to: Parser[A]): this.type =
      derived(this, from, c, to)

    private[parseback] def derived[A](table: MemoTable, from: Parser[A], c: Char, to: Parser[A]): this.type = {
-     derivations.put((table, new ParserId(from)), (c, to))
+     derivations.put((table, new ParserIdentification(from)), (c, to))

      this
    }
@@ -64,7 +64,7 @@ private[parseback] final class InitialMemoTable extends MemoTable {
      derive(this, from, c)

    private[parseback] def derive[A](table: MemoTable, from: Parser[A], c: Char): Option[Parser[A]] = {
-     val back = derivations.get((table, new ParserId(from)))
+     val back = derivations.get((table, new ParserIdentification(from)))

      if (back != null && back._1 == c)
        Some(back._2.asInstanceOf[Parser[A]])
@@ -76,7 +76,7 @@ private[parseback] final class InitialMemoTable extends MemoTable {
      finished(this, target, results)

    private[parseback] def finished[A](table: MemoTable, target: Parser[A], results: Results.Cacheable[A]): this.type = {
-     finishes.put((table, new ParserId(target)), results)
+     finishes.put((table, new ParserIdentification(target)), results)

      this
    }
@@ -85,7 +85,7 @@ private[parseback] final class InitialMemoTable extends MemoTable {
      finish(this, target)

    private[parseback] def finish[A](table: MemoTable, target: Parser[A]): Option[Results.Cacheable[A]] =
-     Option(finishes.get((table, new ParserId(target)))).asInstanceOf[Results.Cacheable[A]])
+     Option(finishes.get((table, new ParserIdentification(target))).asInstanceOf[Results.Cacheable[A]])

    def recreate(): MemoTable = new FieldMemoTable(this)
  }
```

```
diff --git a/core/src/main/scala/parseback/parsers.scala b/core/
↪ src/main/scala/parseback/parsers.scala
index 960404f..40373a7 100644
--- a/core/src/main/scala/parseback/parsers.scala
+++ b/core/src/main/scala/parseback/parsers.scala
@@ -24,7 +24,7 @@ import scala.annotation.unchecked.
↪ uncheckedVariance
 import scala.util.matching.{Regex => SRegex}
 import scala.util.{Either, Left, Right}

-import MemoTable.ParserId
+import MemoTable.{ParserIdentification => ParserId}

 sealed trait Parser[+A] {
```

Listing 5.1: Code Review Scenario 1 Step 0

The second syntactic change is the update of the interface *Memotable* from a *sealed abstract class* to *sealed trait*, as can be seen in Listing 5.2. While theoretically this results in a minor difference how the code is compiled, it should not have any semantic relevance for the resulting library.

```
diff --git a/core/src/main/scala/parseback/MemoTable.scala b/core
↪ /src/main/scala/parseback/MemoTable.scala
index dd2033f..fa70fcc 100644
--- a/core/src/main/scala/parseback/MemoTable.scala
+++ b/core/src/main/scala/parseback/MemoTable.scala
@@ -20,7 +20,7 @@ import java.util.HashMap

 // note that there are only two implementation here, preserving
 ↪ bimorphic PIC
 // TODO it may be possible to retain SOME results between
 ↪ derivations (just not those which involve Apply)
-private[parseback] sealed abstract class MemoTable {
+private[parseback] sealed trait MemoTable {

   def derived[A](from: Parser[A], c: Char, to: Parser[A]): this.
   ↪ type
   def derive[A](from: Parser[A], c: Char): Option[Parser[A]]
```

Listing 5.2: Code Review Scenario 1 Step 1

In the last commit of this scenario a method definition is moved into a different scope. Listing 5.3 is the diff of the commit for this change.

```
diff --git a/core/src/main/scala/parseback/render/Renderer.scala
↪ b/core/src/main/scala/parseback/render/Renderer.scala
index 646cbcc..2855630 100644
```

```
--- a/core/src/main/scala/parseback/render/Renderer.scala
+++ b/core/src/main/scala/parseback/render/Renderer.scala
@@ -33,39 +33,6 @@ object Renderer {

   // graph rendering is complicated... :-/
   final def renderCompact(self: Parser[_]): String = {
-    def renderNonterminal(label: String, target: List[
↪ RenderResult.TokenSequence]): State[RenderState, String] = {
-      require(!target.isEmpty)
-
-      val init = label :: "::=" :: Nil
-
-      def handleSequence(seq: RenderResult.TokenSequence): State
↪ [RenderState, List[String]] =
-        for {
-          st <- State.get[RenderState]
-          (map, _) = st
-
-          inverted = Map(map.toList map {
-            case (label, (target, _)) => target -> label
-          }: _*)
-
-          rendered <- seq traverse {
-            case Left(p) =>
-              if (inverted contains p) {
-                State.pure[RenderState, List[String]](inverted(p
↪ ) :: Nil)
-              } else {
-                render(p) flatMap handleSequence
-              }
-
-            case Right(str) =>
-              State.pure[RenderState, List[String]](str :: Nil)
-          }
-        } yield rendered.flatten
-
-      for {
-        renderedBranches <- target traverse handleSequence
-        rendered = renderedBranches reduce { _ ::: "|" :: _ }
-      } yield (init ::: rendered) mkString " "
-    }
-
     val renderAll = for {
       start <- render(self)

@@ -143,6 +110,39 @@ object Renderer {
       State pure (("!!" :: Nil) map { Right(_) })
   }

+  private def renderNonterminal(label: String, target: List[
↪ RenderResult.TokenSequence]): State[RenderState, String] = {
+    require(!target.isEmpty)
+
+    val init = label :: "::=" :: Nil
+
+    def handleSequence(seq: RenderResult.TokenSequence): State[RenderState, List[String]] =
+      for {
+        st <- State.get[RenderState]
+        (map, _) = st
+
+        inverted = Map(map.toList map {
+          case (label, (target, _)) => target -> label
+        }: _*)
+
+        rendered <- seq traverse {
+          case Left(p) =>
+            if (inverted contains p) {
+              State.pure[RenderState, List[String]](inverted(p) :: Nil)
+            } else {
+              render(p) flatMap handleSequence
+            }
+
+          case Right(str) =>
+            State.pure[RenderState, List[String]](str :: Nil)
+        }
+      } yield rendered.flatten
+
+    for {
+      renderedBranches <- target traverse handleSequence
+      rendered = renderedBranches reduce { _ ::: "|" :: _ }
+    } yield (init ::: rendered) mkString " "
+  }
+
   private def gatherBranches(self: Parser[_], root: Option[Parser[_]]): List[Parser[_]] = self match {
     case self @ Union(_, _) =>
       root match {
```

Listing 5.3: Code Review Scenario 1 Step 2

The goal of this scenario is to test how well *PolyCoDif* can be used to display source code with changes in the interface. To this end, the user is asked if there are any significant semantic changes and if the result, when for example, executing the tests for the library, have changed. A scale between 1 and 10, where 1 is no semantic change at all and 10 is a significant change, is used to measure the end-users answer. The expected answer is 1, but to take into account that end-users might not be familiar with every Scala feature, 4 or lower will be accepted as a positive answer.

CRW The second code review task will focus on a potential **worst-case scenario** for the developed prototype, changes in the body of one specific method. Most changes in the two commits will be reformatting of code style and simple renaming of variables. For example, the function calls in the method *render* will be changed from infix style to usual . notation.

Additionally, there is a significant change in the method *gatherBranches*. This method is a recursive method and for optimization purposes it is rewritten in a tail recursive way. The patches can be found in Section 5.1.6.

The end-user will again be asked if there are any significant semantic changes. The scale is again between 1 and 10 where 1 is no semantic change and 10 is substantial semantic change. Since there is some different behavior in corner cases and because the original method might fail due to *Stack Overflow Exceptions* there is actually some semantic change. Therefore, a value of 5 or lower will be accepted as a positive answer.

Code Review – Scenario 2 The scenario consists of tow distinct changes, which are listed in the following steps. The Listing 5.4 is a commit which converts a function from a recursive one to a tail-recursive method.

```
diff --git a/core/src/main/scala/parseback/render/Renderer.scala
↪ b/core/src/main/scala/parseback/render/Renderer.scala
index 646cbcc..b2c5080 100644
--- a/core/src/main/scala/parseback/render/Renderer.scala
+++ b/core/src/main/scala/parseback/render/Renderer.scala
@@ -17,6 +17,8 @@
 package parseback
 package render

+import scala.annotation.tailrec
+
 import cats.instances.list._
 import cats.instances.option._
 import cats.data.State
@@ -106,21 +108,21 @@ object Renderer {
     case Sequence(left, _, right) =>
       State pure (Left(left) :: Left(right) :: Nil)

-    case Union(_, _) =>
+    case u@Union(_, _) =>
       for {
         st <- State.get[RenderState]
         (nts, labels) = st

-        back <- if (nts.values exists { case (p, _) => p eq self
↪ }) {
-          State.pure[RenderState, RenderResult.TokenSequence](
↪ Left(self) :: Nil)
+        back <- if (nts.values exists { case (p, _) => p eq u })
↪ {
+          State.pure[RenderState, RenderResult.TokenSequence](
↪ Left(u) :: Nil)
         } else {
           val (labels2, label) = assignLabel(labels)

-          val branches = gatherBranches(self, None)
+          val branches = gatherBranches(u :: Nil, Nil)

           for {
-            _ <- State.set((nts + (label -> ((self, branches))),
↪ labels2))
-          } yield Left(self) :: Nil
+               _ <- State.set((nts + (label -> ((u, branches))),
↪ labels2))
+             } yield Left(u) :: Nil
            }
          } yield back

@@ -143,19 +145,12 @@ object Renderer {
          State pure (("!!" :: Nil) map { Right(_) })
      }

-   private def gatherBranches(self: Parser[_], root: Option[
↪ Parser[_]]): List[Parser[_]] = self match {
-     case self @ Union(_, _) =>
-       root match {
-         case Some(p) if p eq self => self :: Nil
-         case Some(_) =>
-           gatherBranches(self.left, root) ::: gatherBranches(
↪ self.right, root)
-
-         case None =>
-           gatherBranches(self.left, Some(self)) :::
↪ gatherBranches(self.right, Some(self))
-       }
-
-     case _ =>
-       self :: Nil
+   @tailrec
+   private def gatherBranches(selfParsers: List[Parser[_]],
↪ result:List[Parser[_]]): List[Parser[_]] = selfParsers match {
+     case (self @ Union(_, _)) :: xs if result contains self =>
↪ gatherBranches(xs, result)
+     case (self @ Union(_, _)) :: xs => gatherBranches(self.left
↪ :: self.right :: xs, result)
+     case self :: xs => gatherBranches(xs, self :: result)
+     case Nil => result
    }

    private final def assignLabel(labels: Set[String]): (Set[
    ↪ String], String) = {
```

Listing 5.4: Code Review Scenario 2 Step 0

The second commit, which can be found in Listing 5.5 changes several infix function calls to the more familiar *dot notation*.

```
diff --git a/core/src/main/scala/parseback/render/Renderer.scala
↪ b/core/src/main/scala/parseback/render/Renderer.scala
index b2c5080..1adc1d5 100644
--- a/core/src/main/scala/parseback/render/Renderer.scala
+++ b/core/src/main/scala/parseback/render/Renderer.scala
@@ -106,7 +106,7 @@ object Renderer {

   def render(self: Parser[_]): State[RenderState, RenderResult.TokenSequence] = self match {
     case Sequence(left, _, right) =>
-      State pure (Left(left) :: Left(right) :: Nil)
+      State.pure(Left(left) :: Left(right) :: Nil)

     case u@Union(_, _) =>
       for {
@@ -127,30 +127,34 @@ object Renderer {
       } yield back

     case Apply(target, _, _) =>
-      State pure (Left(target) :: Right("↪") :: Right("λ") :: Nil)
+      State.pure(Left(target) :: Right("↪") :: Right("λ") :: Nil)

     case Filter(target, _) =>
-      State pure (Left(target) :: Nil)
+      State.pure(Left(target) :: Nil)

     case Literal(literal, offset) =>
-      State pure ((s"'${literal substring offset}'" :: Nil) map { Right(_) })
+      State.pure((s"'${literal substring offset}'" :: Nil) map { Right(_) })

     case Regex(r) =>
-      State pure ((s"/${r.pattern}/" :: Nil) map { Right(_) })
+      State.pure((s"/${r.pattern}/" :: Nil) map { Right(_) })

     case Epsilon(value) =>
-      State pure ((s"ϵ=${value.toString}" :: Nil) map { Right(_) })
+      State.pure((s"ϵ=${value.toString}" :: Nil) map { Right(_) })

     case Failure(errors) =>
-      State pure (("!!" :: Nil) map { Right(_) })
+      State.pure(("!!" :: Nil) map { Right(_) })
   }

   @tailrec
   private def gatherBranches(selfParsers: List[Parser[_]], result:List[Parser[_]]): List[Parser[_]] = selfParsers match {
-    case (self @ Union(_, _)) :: xs if result contains self => gatherBranches(xs, result)
-    case (self @ Union(_, _)) :: xs => gatherBranches(self.left :: self.right :: xs, result)
-    case self :: xs => gatherBranches(xs, self :: result)
-    case Nil => result
+    case (self @ Union(_, _)) :: xs if result contains self =>
+      gatherBranches(xs, result)
+    case (self @ Union(_, _)) :: xs =>
+      gatherBranches(self.left :: self.right :: xs, result)
+    case self :: xs =>
+      gatherBranches(xs, self :: result)
+    case Nil =>
+      result
   }

   private final def assignLabel(labels: Set[String]): (Set[
   ↪ String], String) = {
```

Listing 5.5: Code Review Scenario 2 Step 1

Code Churn

Code churn is defined as a lot of changes to a very localized part of the code. This may happen if a lot of developers are working on similar code parts. There are again two parts to this.

CCB In the **best-case scenario** the user is presented with a lot of change in one file. The changes all revolve around the method *next* in the class *Line*.

In the following commits variations of the *filter* condition are used. During step 2, which can be seen in Listing 5.8, a new method *nonEmpty* is introduced to work around the boolean negation. This is the only method which was introduced during all the refactoring. The goal for the end-user is to find out whether at the end of all the changes, the interface of the class has changed. Every negative answer will be accepted as correct.

The patches can be found in Section 5.1.6.

Code Churn – Scenario 1 To simulate code churn, six individual steps were performed. The git diff for step 0 looks like a new method was introduced, but in reality the *isEmpty* function was only moved. This can be seen in Listing 5.6.

```
diff --git a/core/src/main/scala/parseback/Line.scala b/core/src/
  ↪ main/scala/parseback/Line.scala
index 9415d59..565c40b 100644
--- a/core/src/main/scala/parseback/Line.scala
+++ b/core/src/main/scala/parseback/Line.scala
@@ -24,35 +24,27 @@ package parseback
 final case class Line(base: String, lineNo: Int = 0, colNo: Int
  ↪ = 0) {

   def head: Char = base charAt colNo
+  def isEmpty: Boolean = base.length == colNo

   def project: String = base substring colNo

-  def isEmpty: Boolean = base.length == colNo
-
-  def next: Option[Line] =
-    Some(Line(base, lineNo, colNo + 1)) filter (!_.isEmpty)
```

```
+  def next: Option[Line] = Some(Line(base, lineNo, colNo + 1))
↪ filter (!_.isEmpty)

   def isBefore(that: Line): Boolean =
     this.lineNo < that.lineNo || (this.lineNo == that.lineNo &&
     ↪ this.colNo < that.colNo)

-  def renderError: String =
-    base + s"${0 until colNo map { _ => ' ' } mkString}^"
+  def renderError: String = base + s"${0 until colNo map { _ =>
↪ ' ' } mkString}^"
 }

-object Line extends ((String, Int, Int) => Line) {
-
+object Line {
   def addTo(lines: Vector[Line], line: Line): Vector[Line] = {
-    if (lines.isEmpty) {
+    if (lines.isEmpty)
       Vector(line)
-    } else {
-      val last = lines.last
-
-      if (last.lineNo < line.lineNo)
-        lines :+ line
-      else if (lines.length == 1)
-        lines
-      else
-        lines.updated(lines.length - 1, line)
-    }
+   else if (lines.last.lineNo < line.lineNo)
+     lines :+ line
+   else if (lines.length == 1)
+     lines
+   else
+     lines.updated(lines.length - 1, line)
   }
 }
```

Listing 5.6: Code Churn Scenario 1 Step 0

The Listing 5.7 represents a conversion from infix to *dot notation*.

```
diff --git a/core/src/main/scala/parseback/Line.scala b/core/src/
↪ main/scala/parseback/Line.scala
index 565c40b..3a0ec83 100644
--- a/core/src/main/scala/parseback/Line.scala
+++ b/core/src/main/scala/parseback/Line.scala
@@ -22,13 +22,11 @@ package parseback
   * @param colNo The column offset into `base` (0 indexed)
   */
```

```
final case class Line(base: String, lineNo: Int = 0, colNo: Int = 0) {
-
  def head: Char = base charAt colNo
  def isEmpty: Boolean = base.length == colNo
-
  def project: String = base substring colNo

- def next: Option[Line] = Some(Line(base, lineNo, colNo + 1)) filter (!_.isEmpty)
+ def next: Option[Line] = Some(Line(base, lineNo, colNo + 1)).filter(!_.isEmpty)

  def isBefore(that: Line): Boolean =
    this.lineNo < that.lineNo || (this.lineNo == that.lineNo && this.colNo < that.colNo)
```

Listing 5.7: Code Churn Scenario 1 Step 1

The Listing 5.8 is the aforementioned introduction of the new method.

```
diff --git a/core/src/main/scala/parseback/Line.scala b/core/src/main/scala/parseback/Line.scala
index 3a0ec83..abaa9fe 100644
--- a/core/src/main/scala/parseback/Line.scala
+++ b/core/src/main/scala/parseback/Line.scala
@@ -24,6 +24,7 @@ package parseback
 final case class Line(base: String, lineNo: Int = 0, colNo: Int = 0) {
  def head: Char = base charAt colNo
  def isEmpty: Boolean = base.length == colNo
+ private def nonEmpty = !isEmpty
  def project: String = base substring colNo

  def next: Option[Line] = Some(Line(base, lineNo, colNo + 1)).filter(!_.isEmpty)
```

Listing 5.8: Code Churn Scenario 1 Step 2

Listing 5.9 introduces a usage of this method.

```
diff --git a/core/src/main/scala/parseback/Line.scala b/core/src/main/scala/parseback/Line.scala
index abaa9fe..6f4e365 100644
--- a/core/src/main/scala/parseback/Line.scala
+++ b/core/src/main/scala/parseback/Line.scala
@@ -27,7 +27,7 @@ final case class Line(base: String, lineNo: Int = 0, colNo: Int = 0) {
   private def nonEmpty = !isEmpty
   def project: String = base substring colNo

-  def next: Option[Line] = Some(Line(base, lineNo, colNo + 1)).
 ↪ filter(!_.isEmpty)
+  def next: Option[Line] = Some(Line(base, lineNo, colNo + 1)).
 ↪ filter(_.nonEmpty)

   def isBefore(that: Line): Boolean =
     this.lineNo < that.lineNo || (this.lineNo == that.lineNo &&
     ↪ this.colNo < that.colNo)
```

Listing 5.9: Code Churn Scenario 1 Step 3

Listing 5.10 changes the visibility of this method to *public*.

```
diff --git a/core/src/main/scala/parseback/Line.scala b/core/src/
 ↪ main/scala/parseback/Line.scala
index 6f4e365..6e23d94 100644
--- a/core/src/main/scala/parseback/Line.scala
+++ b/core/src/main/scala/parseback/Line.scala
@@ -24,7 +24,7 @@ package parseback
 final case class Line(base: String, lineNo: Int = 0, colNo: Int
  ↪ = 0) {
   def head: Char = base charAt colNo
   def isEmpty: Boolean = base.length == colNo
-  private def nonEmpty = !isEmpty
+  def nonEmpty = !isEmpty
   def project: String = base substring colNo

   def next: Option[Line] = Some(Line(base, lineNo, colNo + 1)).
   ↪ filter(_.nonEmpty)
```

Listing 5.10: Code Churn Scenario 1 Step 4

The last step, Listing 5.11, makes the newly introduced method unnecessary and removes it again.

```
diff --git a/core/src/main/scala/parseback/Line.scala b/core/src/
 ↪ main/scala/parseback/Line.scala
index 6e23d94..7ed4d40 100644
--- a/core/src/main/scala/parseback/Line.scala
+++ b/core/src/main/scala/parseback/Line.scala
@@ -24,10 +24,9 @@ package parseback
 final case class Line(base: String, lineNo: Int = 0, colNo: Int
  ↪ = 0) {
   def head: Char = base charAt colNo
   def isEmpty: Boolean = base.length == colNo
-  def nonEmpty = !isEmpty
```

```
   def project: String = base substring colNo

-  def next: Option[Line] = Some(Line(base, lineNo, colNo + 1)).
↪ filter(_.nonEmpty)
+  def next: Option[Line] = Some(Line(base, lineNo, colNo + 1)).
↪ filterNot(_.isEmpty)

   def isBefore(that: Line): Boolean =
     this.lineNo < that.lineNo || (this.lineNo == that.lineNo &&
     ↪ this.colNo < that.colNo)
```

Listing 5.11: Code Churn Scenario 1 Step 5

CCW On the other hand in the **worst-case scenario** a single file is changed continuously in different styles to do performance optimizations. The project will contain 5 commits. The patches can be found in Section 5.1.6. Each commit is going to significantly change the implementation, while keeping the semantics and the interface stable. The code is refactored from a pattern matching style to a direct representation of a ternary logic via integers. The goal for the end-user is to describe if the interface was stable. Every affirmative answer will be accepted as a positive answer.

Code Churn – Scenario 2 The worst case scenario consists of seven individual commits. Each individual commit explores a different way to encode a ternary boolean logic. The commit in Listing 5.12 converts the integer based encoding to an *Algebraic Data Type* encoded with a *sealed trait* and three *case objects*. The boolean operations *and* and *or* are directly implemented in the *case objects* via dynamic dispatch.

```
diff --git a/core/src/main/scala/parseback/Nullable.scala b/core/
↪ src/main/scala/parseback/Nullable.scala
index f885ac8..40f4806 100644
--- a/core/src/main/scala/parseback/Nullable.scala
+++ b/core/src/main/scala/parseback/Nullable.scala
@@ -16,37 +16,28 @@

 package parseback

-import Nullable.{ False, Maybe, True }

 // a Kleene algebra
-private[parseback] final class Nullable(val value: Byte) extends
↪ AnyVal {
-  def ||(that: Nullable): Nullable = (this: @unchecked) match {
-    case True => True
-    case Maybe => if (that == True) True else Maybe
-    case False => that
-  }
+private[parseback] sealed trait Nullable {
+  def ||(that: Nullable): Nullable
+  def &&(that: Nullable): Nullable
+  def toBoolean: Boolean
+}
```

```

-  def &&(that: Nullable): Nullable = (this: @unchecked) match {
-    case True  => that
-    case Maybe => if (that == False) False else Maybe
-    case False => False
+private[parseback] object Nullable {
+  case object True extends Nullable {
+    def ||(that: Nullable): Nullable = True
+    def &&(that: Nullable): Nullable = that
+    def toBoolean: Boolean = true
   }
-
-  def toBoolean: Boolean = (this: @unchecked) match {
-    case True  => true
-    case Maybe => sys.error("not intended to be called")
-    case False => false
+  case object Maybe extends Nullable {
+    def ||(that: Nullable): Nullable = if (that == True) True
↪ else Maybe
+    def &&(that: Nullable): Nullable = if (that == False) False
↪ else Maybe
+    def toBoolean: Boolean = sys.error("not intended to be
↪ called")
   }
-
-  override def toString = (this: @unchecked) match {
-    case True  => "True"
-    case Maybe => "Maybe"
-    case False => "False"
+  case object False extends Nullable {
+    def ||(that: Nullable): Nullable = that
+    def &&(that: Nullable): Nullable = False
+    def toBoolean: Boolean = false
   }
 }
-
-private[parseback] object Nullable {
-  val True  = new Nullable(1)
-  val Maybe = new Nullable(0)
-  val False = new Nullable(-1)
-}
```

Listing 5.12: Code Churn Scenario 2 Step 0

In the next step, represented by Listing 5.13, moves the implementation of these boolean methods from the *case objects* to the parent *sealed trait* and relies on *pattern matching*.

```
diff --git a/core/src/main/scala/parseback/Nullable.scala b/core/
↪ src/main/scala/parseback/Nullable.scala
index 40f4806..a3c1ecb 100644
--- a/core/src/main/scala/parseback/Nullable.scala
```

```
+++ b/core/src/main/scala/parseback/Nullable.scala
@@ -16,28 +16,29 @@

 package parseback

+import Nullable.{ False, Maybe, True }

 // a Kleene algebra
-private[parseback] sealed trait Nullable {
-  def ||(that: Nullable): Nullable
-  def &&(that: Nullable): Nullable
-  def toBoolean: Boolean
-}
+private[parseback] final class Nullable(private val value: Int) extends AnyVal {
+  def ||(that: Nullable): Nullable = new Nullable(this.value & that.value)

-private[parseback] object Nullable {
-  case object True extends Nullable {
-    def ||(that: Nullable): Nullable = True
-    def &&(that: Nullable): Nullable = that
-    def toBoolean: Boolean = true
-  }
-  case object Maybe extends Nullable {
-    def ||(that: Nullable): Nullable = if (that == True) True else Maybe
-    def &&(that: Nullable): Nullable = if (that == False) False else Maybe
-    def toBoolean: Boolean = sys.error("not intended to be called")
+  def &&(that: Nullable): Nullable = new Nullable(this.value | that.value)
+
+  def toBoolean: Boolean = (this: @unchecked) match {
+    case True  => true
+    case Maybe => sys.error("not intended to be called")
+    case False => false
   }
-  case object False extends Nullable {
-    def ||(that: Nullable): Nullable = that
-    def &&(that: Nullable): Nullable = False
-    def toBoolean: Boolean = false
+
+  override def toString = (this: @unchecked) match {
+    case True  => "True"
+    case Maybe => "Maybe"
+    case False => "False"
   }
 }
+
+private[parseback] object Nullable {
+  val True  = new Nullable(0)
+  val Maybe = new Nullable(1)
+  val False = new Nullable(-1)
+}
```

Listing 5.13: Code Churn Scenario 2 Step 1

The Listing 5.14 is a commit which changes the encoding back to an integer based one. This encoding is similar to the initial one, but is slightly more efficient.

```
diff --git a/core/src/main/scala/parseback/Nullable.scala b/core/
↪ src/main/scala/parseback/Nullable.scala
index a3c1ecb..6f9ad08 100644
--- a/core/src/main/scala/parseback/Nullable.scala
+++ b/core/src/main/scala/parseback/Nullable.scala
@@ -24,16 +24,16 @@ private[parseback] final class Nullable(
↪ private val value: Int) extends AnyVal {

   def &&(that: Nullable): Nullable = new Nullable(this.value |
   ↪ that.value)

-  def toBoolean: Boolean = (this: @unchecked) match {
-    case True  => true
-    case Maybe => sys.error("not intended to be called")
-    case False => false
+  def toBoolean: Boolean = this.value match {
+    case 0  => true
+    case 1 => sys.error("not intended to be called")
+    case -1 => false
   }

-  override def toString = (this: @unchecked) match {
-    case True  => "True"
-    case Maybe => "Maybe"
-    case False => "False"
+  override def toString = this.value match {
+    case 0  => "True"
+    case 1 => "Maybe"
+    case -1 => "False"
   }
 }
```

Listing 5.14: Code Churn Scenario 2 Step 2

The Listing 5.14, 5.15, 5.16, 5.17 and 5.18 are only minor variations in formatting, cleanups with regards to imports and relying less on magic numbers.

```
diff --git a/core/src/main/scala/parseback/Nullable.scala b/core/
↪ src/main/scala/parseback/Nullable.scala
index 6f9ad08..4047357 100644
```

```
--- a/core/src/main/scala/parseback/Nullable.scala
+++ b/core/src/main/scala/parseback/Nullable.scala
@@ -16,8 +16,6 @@

 package parseback

-import Nullable.{ False, Maybe, True }
-
 // a Kleene algebra
 private[parseback] final class Nullable(private val value: Int)
 ↪ extends AnyVal {
   def ||(that: Nullable): Nullable = new Nullable(this.value &
   ↪ that.value)
```

Listing 5.15: Code Churn Scenario 2 Step 3

```
diff --git a/core/src/main/scala/parseback/Nullable.scala b/core/
↪ src/main/scala/parseback/Nullable.scala
index 4047357..c35cd4a 100644
--- a/core/src/main/scala/parseback/Nullable.scala
+++ b/core/src/main/scala/parseback/Nullable.scala
@@ -23,15 +23,15 @@ private[parseback] final class Nullable(
↪ private val value: Int) extends AnyVal {
   def &&(that: Nullable): Nullable = new Nullable(this.value |
   ↪ that.value)

   def toBoolean: Boolean = this.value match {
-    case 0  => true
-    case 1 => sys.error("not intended to be called")
-    case -1 => false
+    case  0  => true
+    case  1  => sys.error("not intended to be called")
+    case -1  => false
   }

   override def toString = this.value match {
-    case 0  => "True"
-    case 1 => "Maybe"
-    case -1 => "False"
+    case  0  => "True"
+    case  1  => "Maybe"
+    case -1  => "False"
   }
 }
```

Listing 5.16: Code Churn Scenario 2 Step 4

```
diff --git a/core/src/main/scala/parseback/Nullable.scala b/core/
↪ src/main/scala/parseback/Nullable.scala
index c35cd4a..93870cf 100644
--- a/core/src/main/scala/parseback/Nullable.scala
+++ b/core/src/main/scala/parseback/Nullable.scala
@@ -16,6 +16,8 @@

 package parseback

+import Nullable.{ False, Maybe, True }
+
 // a Kleene algebra
 private[parseback] final class Nullable(private val value: Int)
 ↪ extends AnyVal {
   def ||(that: Nullable): Nullable = new Nullable(this.value &
   ↪ that.value)
@@ -23,15 +25,15 @@ private[parseback] final class Nullable(
↪ private val value: Int) extends AnyVal {
   def &&(that: Nullable): Nullable = new Nullable(this.value |
   ↪ that.value)

   def toBoolean: Boolean = this.value match {
-    case  0 => true
-    case  1 => sys.error("not intended to be called")
-    case -1 => false
+    case True.value  => true
+    case Maybe.value => sys.error("not intended to be called")
+    case False.value => false
   }

   override def toString = this.value match {
-    case  0 => "True"
-    case  1 => "Maybe"
-    case -1 => "False"
+    case True.value  => "True"
+    case Maybe.value => "Maybe"
+    case False.value => "False"
   }
 }
```

Listing 5.17: Code Churn Scenario 2 Step 5

```
diff --git a/core/src/main/scala/parseback/Nullable.scala b/core/
↪ src/main/scala/parseback/Nullable.scala
index 93870cf..ba58703 100644
--- a/core/src/main/scala/parseback/Nullable.scala
+++ b/core/src/main/scala/parseback/Nullable.scala
@@ -38,7 +38,7 @@ private[parseback] final class Nullable(private
↪  val value: Int) extends AnyVal {
 }

 private[parseback] object Nullable {
```

```
- val True  = new Nullable(0)
- val Maybe = new Nullable(1)
- val False = new Nullable(-1)
+ final val True  = new Nullable(0)
+ final val Maybe = new Nullable(1)
+ final val False = new Nullable(-1)
  }
```

Listing 5.18: Code Churn Scenario 2 Step 6

5.2 Results

Over the course of several weeks ten participants were chosen to take part in the scenario-based expert evaluation. To increase the reach of the scenario-based expert evaluation it was possible for the participants to choose whether to participate in person or remote. The procedure was nearly identical in both cases.

Of the ten original test subjects eight answered the questionnaire. From these eight six proofed useful for this thesis.

The reasons for the two invalid results were in the one case an incomplete filled out questionnaire and in the other case the need to abort the scenario-based expert evaluation due to time constraints from the participant.

In the following sections an overview of the results is given.

5.2.1 Demographics

The demographics section will only focus on the six people whose results are usable for the evaluation of the prototype. Of these six participants all identified as male and they were between 27 and 32 years old. They are working in the area of software engineering in different roles. The participants were all people experienced in software engineering but with different levels of expertise.

With the sentence „I have a lot of experience developing software“ two participants strongly agree, one participant agreed and 3 participants neither agreed nor disagreed. The participants where less skilled in reviewing code. With the sentence „I have a lot of experience reviewing code“ only one person answered strongly agreed while the rest of the answers ranged from disagreeing to agreeing. The detailed results can be found in Table 5.1 and visualized in Figure 5.2.

	Strongly Disagree				Strongly Agree
I have a lot of experience developing software	0	0	3	1	2
I have a lot of experience reviewing code	0	1	2	2	1
With **Scala**	2	0	1	1	2
With **Java**	0	1	2	1	2
With **Javascript**	4	0	2	0	0

Table 5.1: Results for the experience questions.

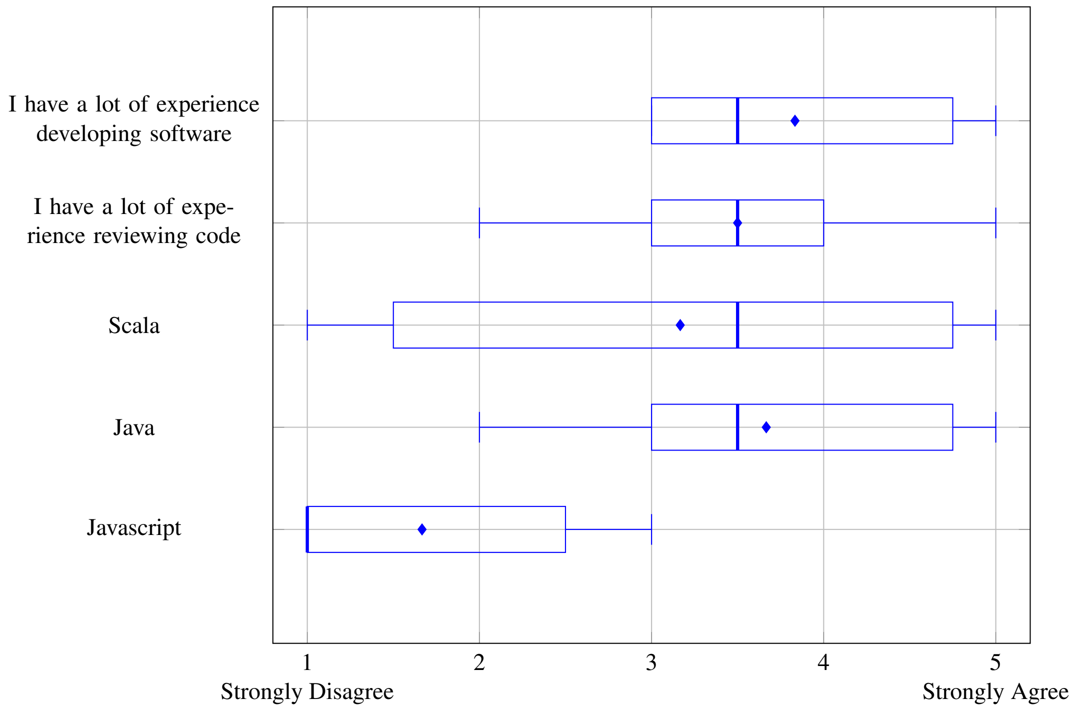


Figure 5.2: Demographics of the participants

In addition the question was asked which tools the participants prefer for writing and reviewing code.

IntelliJ IDEA	Eclipse	Visual Studio	VSCode	other:
5	0	1	1	0

Table 5.2: My preferred tool to write code is

IntelliJ IDEA	GitHub	git command	BitBucket	other:
4	0	1	3	0

Table 5.3: My preferred tool to review code is

There was a strong dominance of IntelliJ IDEA both while reviewing as well as writing code.

5.2.2 Task results

In this section each scenario will be evaluated independently. In Chapter 6 the results will be interpreted and a connection to the research questions of the thesis will be made.

Code Review

The results for the two tasks in the code review scenario can be found in Figures 5.3 and 5.4.

According to the limits set in the description three out of the six participants completed the first task successfully and 2 out of the six participants answered the second task correctly.

Of note here is mostly the different meaning the participants of the scenario-based expert evaluation ascribed to the meaning of the word „significant“.

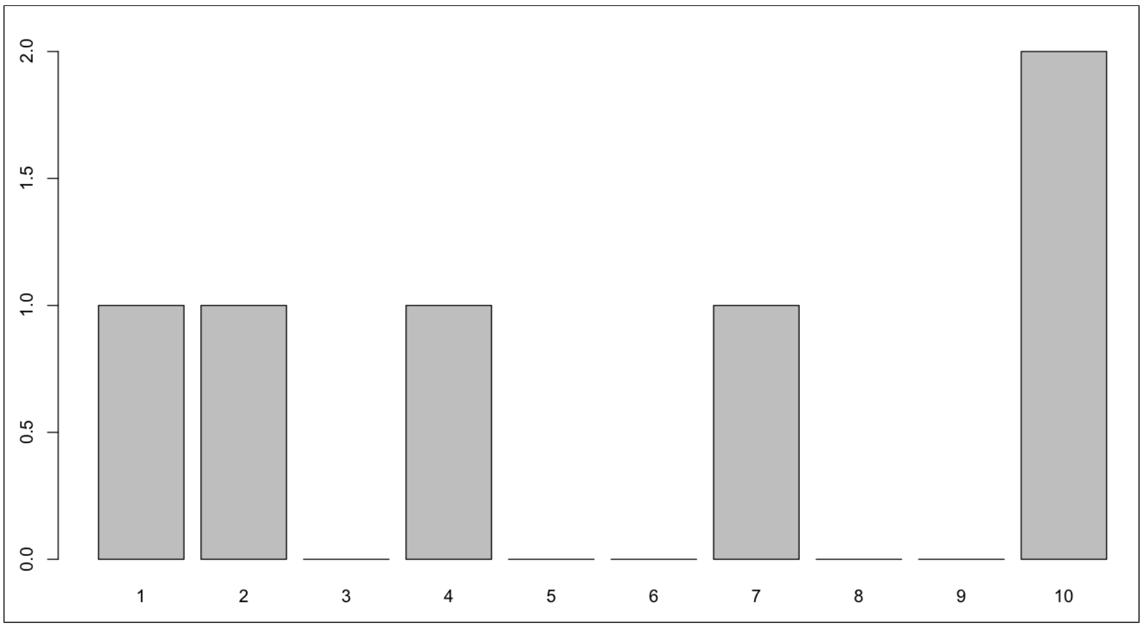


Figure 5.3: The results of the CRB task. The x-axis represents the options presented in the questionnaire, the y-axis is the amount of persons which chose this option

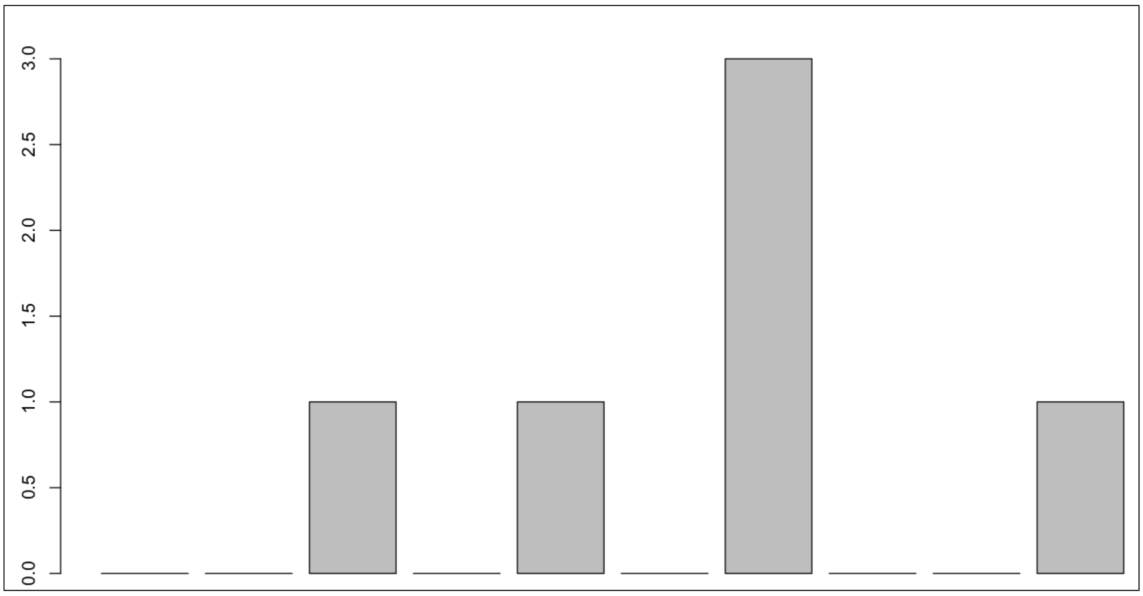


Figure 5.4: The results of the CRW task. The x-axis represents the options presented in the questionnaire, the y-axis is the amount of persons which chose this option

During the discussions at the end of the scenario-based expert evaluation it was made clear that the three test subjects who answered correctly only considered a semantic change as „significant“ while the other half determined from the amount of changes if it was significant or not.

For the worst-case scenario the results were a bit worse as expected. Only two out of the six testees gave correct answers.

The questions about the perceived usefulness of the tool where positive. Three participants answered that they agreed with the statement „I found the system was useful to answer the question for this scenario.“ and two participant answered that they would use the tool in similar scenarios in the future.

Code Churn

For the code churn scenario the first task was overwhelmingly successful. The question here was focused on the interface of a class and if, after 4 commits the definitions of the class where the same as at the beginning. Every participant of the user-study could answer the question. And the general opinion was that *PolyCoDif* was very useful to solve the scenario.

For the worst-case task in the code churn scenario the results where more split. Three participants answered that *yes*, there were new methods introduced and three participants answered with *no*. In general the testees found the prototype again useful to answer the questions.

	Strongly Disagree				Strongly Agree
I found the system was useful to answer the question for this scenario.	0	2	2	2	6
I would consider using the system in the future if I encounter a similar scenario.	0	2	3	4	3

Table 5.4: Results for the Satisfaction with the Tool.

And as one can see in the results depicted in Table 5.4 the overwhelming amount of participants would consider the system for similar scenarios in the future.

5.2.3 System Usability Scale Results

As described in the previous section the SUS questionnaire was used to evaluate the general usability of the system.

A result of 68 is considered above average and therefore a good result. PolyCoDif scored a mean value of 73,75 and can therefore be considered slightly above average according to Sauro and Lewis [49].

A detailed box plot of the result can be found in Figure 5.5. An Overview of the final result is pictured in Figure 5.6. In this plot one can see that the worst result was 65 while the best result was a 82,5. In general there were no outliers and even the 25th and 75th percentiles are fairly close together hinting at a high-level of satisfaction among the test subjects with regards to the usability of the prototype.

5.3 Threats to validity

This section discusses the validity of the results derived from the scenario-based expert evaluation.

5.3.1 Number of Participants

Less than 10 participants for the scenario-based expert evaluation is not a large enough pool of users to deliver sufficient quantitative metrics. However, for a first evaluation of a prototype the qualitative approach to user testing should be sufficient. As a more extensive evaluation would go beyond the scope for a master's thesis. Conducting a more thorough evaluation with more participants in a follow up thesis or paper will help obtaining more insights and showing the usefulness of the developed software.

5.3.2 Selection Bias

Since the participants where not selected randomly but are volunteers from the university and coworkers there are no guarantees if the test candidates represent the more general population of software engineers. More thorough background information could be useful in future studies to eliminate any biases.

5.3.3 Manual

To ease the introduction to the prototype, participants where provided a one page-manual for the test sessions. In future studies it might be beneficial to not provide a manual to get less biased and more pristine reactions.

5.3.4 Task Timing

A very useful metric to check for the provided efficiency of the prototype would be time taken to fulfill certain tasks. Unfortunately, the way the experiments where designed this was not easily measurable. For the future studies should be designed to more easily accommodate such metrics.

5.3.5 Task Wording

Some participants of the scenario-based expert evaluation had a lot of follow up questions, which were deliberately not answered by the moderator. Although this was by design it shows that some tasks where confusingly worded. Nevertheless, all participants successfully finished the scenarios which should be proof enough to show that the wordings where sufficiently precise.

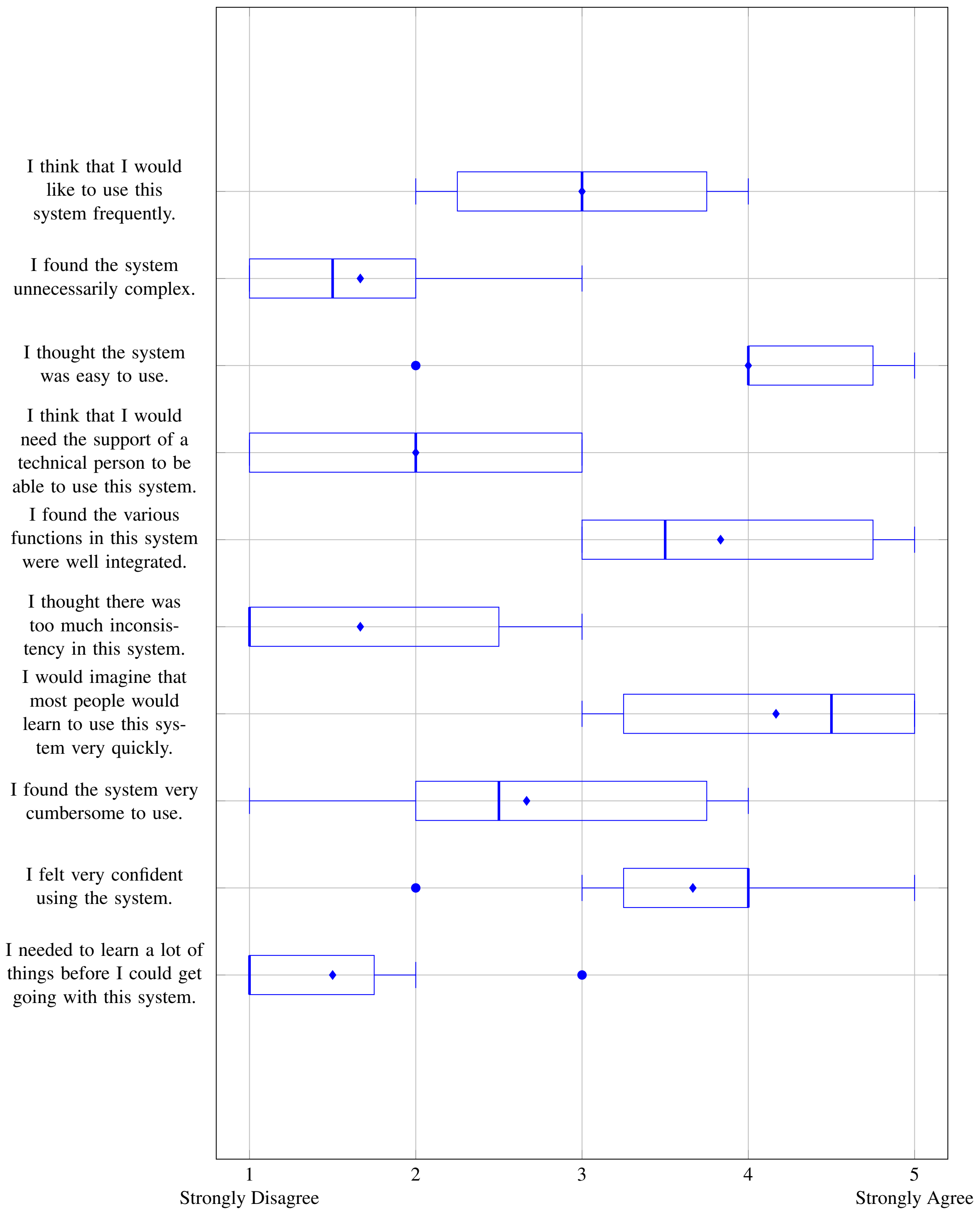


Figure 5.5: System Usability Scale Results

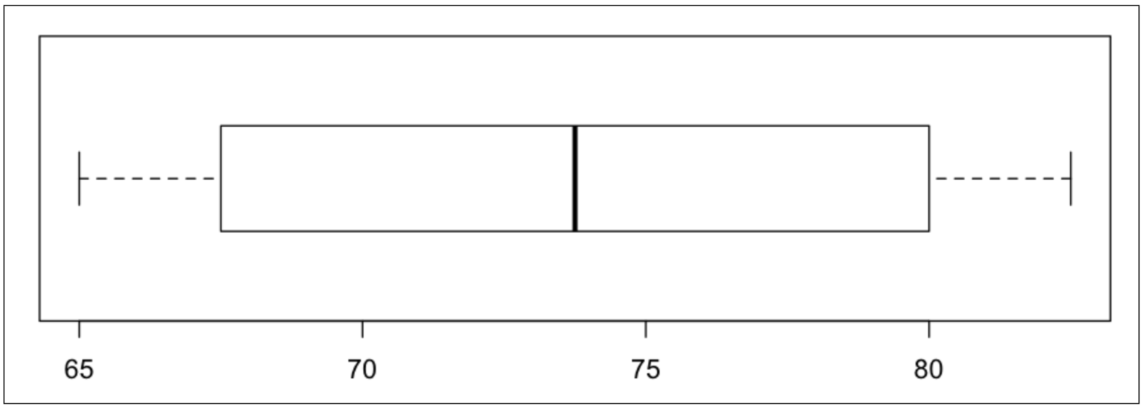


Figure 5.6: The result of the usability evaluation in box plot form.

6 Discussion

In this chapter the thesis attempts to answer the research questions given in Section 1. First the questions will be repeated and afterwards each individual question will be answered.

1. How to combine a tree differencing algorithm with continuous change tracking on top of a polyglot abstract syntax tree?
2. What advantages provides a difference viewer based on a continuous, AST change detection for developers?
3. How do developers rate the usability of a difference viewer based on continuous AST change detection?

6.1 Research Question 1

The first question deals with the theoretical possibilities of combining an algorithm for calculating the edit distance between two trees with a continuous approach. The novelty here is the fact that the tree was independent from a specific programming language but worked in a polyglot setting.

To answer this question an existing algorithm was adapted to work on a generic AST. The prototype started with the canonical implementation of Change Distiller algorithm by Fluri et al. [19]. The implementation originally only worked with Java source code and was extended to recognize and work with code written in Scala.

To verify the success of this approach, the prototype was used on open source software like the parseback project, which is written in Scala, and the PolyCoDif prototype itself, which is written in both Java and Scala.

The approach was tested further by conducting a scenario-based expert evaluation. The results of the SUS questionnaire can further be seen as a proof that the prototype is perceived as a tool with a good user experience and usability.

6.2 Research Question 2

To answer the question whether a difference viewer based on continuous, AST change detection provides advantages for developers a scenario-based expert evaluation was conducted.

This scenario-based expert evaluation focused on the usability of the prototype but in turn neglected the polyglot aspect of the approach. Nevertheless, this more focused scenarios give a good first insight in which cases software engineers felt that the semantic change detection made it easier to answer questions about the given scenarios.

As can be seen in the evaluation section the participants of the scenario-based expert evaluation had especially not trouble to identify whether there were any new methods, variables or constants introduced in the code base. Therefore, one major advantage of PolyCoDif is to review changes on interface level additions and deletions.

Since the separation of interface and implementation, and following that a clear modularization of the software project, is one of the core pillars of both object-oriented and functional programming, PolyCoDif is especially useful for professions like software architects and team leads.

To a minor extend the software is useful when reviewing changes which are the results of a developer refactoring code. In practice and during the scenario-based expert evaluation the participants had again no trouble identifying the high-level changes but due to the nature of the refactoring there were changes in the implementation details as well. These low-level changes where confusing for the testees which lead to incorrect answers given by half of the participants.

The question about the benefits for developers form a difference viewer based on confusing AST tracking was the most difficult to answer. It could be observed that the participants in the scenario-based expert evaluation interacted very little with the timeline slider once they set it to a specific range. Nevertheless, it has the potential to render discussions, how to track changes leading up to the mergeable commit, mute.

For the purposes of this thesis the benefits for end-users where not directly visible. Indirectly though the continuous change tracking forced to build the *PolyCoDif* prototype on a well understood foundation. These algorithms based on patch theory led to a very good performance and therefore to higher scores in the SUS part of the scenario-based expert evaluation.

6.3 Research Question 3

The usability of PolyCoDif was rated above average based on the SUS. Due to the prototype status of the application, bugs were uncovered during the test procedure which might have resulted in a lower score. With proper quality assurance expected from a production ready software, an even higher rating might be possible.

6.4 Conclusion

In summary, one can clearly see that it is possible to combine the existing algorithms in polyglot tree differencing with continuous change tracking. Additionally, in specialized scenarios like refactoring end users felt like they had an advantage in using the prototype. On the other hand the tool lacks in scenarios when the changes where confined to a local environment, e.g. within one method, without touching the interface.

7 Future work

This thesis barely scratched the surface of the possibilities of semantic code differences. There are a lot of opportunities for future research in this area.

7.1 User Interface and Diff Visualization

The most important aspect for adoption in an industrial setting would be in depth research on the visualization of the code changes. In the course of this master thesis two visualizations were developed and one evaluated by a number of participants in a scenario-based expert evaluation. The results of this scenario-based expert evaluation have shown, that the participants would like to use a tool similar to PolyCoDif in the future but that the UI was lacking to correctly answer some of the scenarios.

A collaboration with experts in user experience could proof to be very fruitful and lead to even better outcomes in terms of scenario-based expert evaluation results.

Alternatively, a scenario-based expert evaluation comparing two or more tools could also yield valuable insights for developing the next generation of semantic diff tools.

7.2 Integration with Other Diff Tools

Basing the core of PolyCoDif on the sound principals developed by Mimram and Di Giusto [39] proofed to be necessary to develop a correct tool. The integration with Git on the other hand was at times unnatural since the internal workings of Git are not based on the patches but on snapshots of the history in the repository.

An integration of PolyCoDif with the next generation of VCSs could make this even more natural and pain free. Two examples of such tools would be Darcs and Pijul.

7.3 Continuously Gathered Code Changes

Gathering each individual change done by developers and storing it as a part of the code repository could have the potential to make the debate mute about the size of commits and whether rebasing and squashing of commits are desirable operations or not. However, overall the state of the art in this area seems to be lacking. Very few researchers have been working in this area and as far as the review of the literature has shown, no one is currently concerned about potentials of abuse for this kind of data.

7.4 Improvements on the Generation of Semantic Structure

The implementation of analyzers for Java and Scala as libraries was a good choice for the quick and rapid development of a prototype for PolyCoDif. However, to lift the tool to a industry level state of maturity it should be possible to add new programming languages with less effort.

One way to achieve this would be to take the structure generated by parsers for code highlighting. Two notable implementations would be the parser and symbol indexing engine implemented by the editor Sublime and the open source project TreeSitter which is the next generation parser engine used in the Atom editor.

Alternatively, albeit limited to a specific tool, the ASTs generated by IntelliJ IDEA could be used for the basis of the diff algorithm.

8 Summary

This thesis tried to advance the current state of the art in diff viewers for source code. To achieve this, a prototype was developed which combines the best algorithms in calculating tree-based edit-scripts with the complexity of multi-language tooling in a continuous setting. The three research questions:

- Is there a way to combine a tree differencing algorithm with continuous change tracking on top of a polyglot abstract syntax tree?
- Does a difference viewer based on a continuous, AST change detection provide advantages for developers?
- How do developers benefit from a difference viewer based on continuous AST change detection?

guided the overall implementation and evaluation of the prototype and to correctly answer these questions a scenario-based expert evaluation was conducted.

During the implementation an iterative approach was taken, adding more and more features to *PolyCoDif* which cumulated in a tool with the capabilities of visualizing the differences between two versions of source code based on the structure of the source code. In addition the continuous tracking approach has proven invaluable to develop a sound system which responds very fast to user input, despite the computational complexity of tree-based diff algorithms.

The intuitions about the usability of the prototype have been evaluated and tested in a qualitative survey. In several intensive testing sessions participants needed to answer questions about four different scenarios. Two scenarios dealing with classic code review tasks and two scenarios representing code churn.

The participants finally evaluated the prototype based on the standardized SUS questionnaire. In the discussion section the results of the scenario-based expert evaluation where analysed so that they can function as a basis for future improvements in the area of tree-based edit-scripts and especially the visualizations of such. Based on this a set of opportunities for future work was identified and while there are still a lot of areas where semantic code diff tools in general and *PolyCoDif* in particular can be improved the usefulness and benefits for software engineers can be clearly seen.

Bibliography

References

[1] Apache Software Foundation. *Apache Subversion*. Version 1.9.3. Dec. 15, 2015. URL: http://subversion.apache.org/.

[2] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. „A differencing algorithm for object-oriented programs“. In: *Proceedings of the 19th IEEE international conference on Automated software engineering*. IEEE Computer Society. 2004, pp. 2–13.

[3] David L Atkins et al. „Using version control data to evaluate the impact of software tools: A case study of the version editor“. In: *Software Engineering, IEEE Transactions on* 28.7 (2002), pp. 625–637.

[4] Ira D Baxter et al. „Clone detection using abstract syntax trees“. In: *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE. 1998, pp. 368–377.

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

[6] Lasse Bergroth, Harri Hakonen, and Timo Raita. „A survey of longest common subsequence algorithms“. In: *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*. IEEE. 2000, pp. 39–48.

[7] Philip Bille. „A survey on tree edit distance and related problems“. In: *Theoretical computer science* 337.1-3 (2005), pp. 217–239.

[8] John Brooke et al. „SUS-A quick and dirty usability scale“. In: *Usability evaluation in industry* 189.194 (1996), pp. 4–7.

[9] Eugene Burmako. „SemanticDB: a common data model for Scala developer tools (invited talk)“. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection*. ACM. 2018, pp. 2–2.

[10] Eugene Burmako et al. *Scalameta*. 2004. URL: https://scalameta.org/.

[11] Scott Chacon. *Pro Git*. 1st. Berkely, CA, USA: Apress, 2009. ISBN: 1430218339, 9781430218333.

[12] Sudarshan S Chawathe and Hector Garcia-Molina. „Meaningful change detection in structured data“. In: *ACM SIGMOD Record*. Vol. 26. 2. ACM. 1997, pp. 26–37.

[13] Sudarshan S Chawathe et al. „Change detection in hierarchically structured information“. In: *ACM SIGMOD Record*. Vol. 25. 2. ACM. 1996, pp. 493–504.

[14] Koen Claessen and John Hughes. „QuickCheck: a lightweight tool for random testing of Haskell programs“. In: *Acm sigplan notices* 46.4 (2011), pp. 53–64.

[15] Baojiang Cui et al. „Code comparison system based on abstract syntax tree“. In: *Broadband Network and Multimedia Technology (IC-BNMT), 2010 3rd IEEE International Conference on*. IEEE. 2010, pp. 668–673.

[16] Sébastien Doeraene. „Scala.js: Type-Directed Interoperability with Dynamically Typed Languages“. In: (2013), p. 10.

[17] Roy Fielding. „Representational state transfer“. In: *Architectural Styles and the Design of Netowork-based Software Architecture* (2000), pp. 76–85.

[18] Hans-Christian Fjeldberg. „Polyglot programming. A business perspective“. In: (2008).

[19] Beat Fluri et al. „Change distilling: Tree differencing for fine-grained source code change extraction“. In: *Software Engineering, IEEE Transactions on* 33.11 (2007), pp. 725–743.

[20] Veit Frick, Christoph Wedenig, and Martin Pinzger. „DiffViz: A Diff Algorithm Independent Visualization Tool for Edit Scripts“. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2018, pp. 705–709.

[21] Thomas Fritz and Gail C Murphy. „Using information fragments to answer the questions developers ask“. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM. 2010, pp. 175–184.

[22] *git-notes - Add or inspect object notes*. 2018. URL: https://git-scm.com/docs/git-notes.

[23] Junio C. Hamano and Linus Torvalds. *Git*. Version 2.8.1. Apr. 3, 2016. URL: https://git-scm.com/.

[24] Lile Hattori. „Enhancing collaboration of multi-developer projects with synchronous changes“. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM. 2010, pp. 377–380.

[25] Lile Hattori and Michele Lanza. „Syde: A tool for collaborative software development“. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM. 2010, pp. 235–238.

[26] Lile Hattori, Mircea Lungu, and Michele Lanza. „Replaying past changes in multi-developer projects“. In: *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*. ACM. 2010, pp. 13–22.

[27] Lile Hattori et al. „Answering software evolution questions: An empirical evaluation“. In: *Information and Software Technology* 55.4 (2013), pp. 755–775.

[28] Paul Hudak et al. „A history of Haskell: being lazy with class“. In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM. 2007, pp. 12–1.

[29] Judah Jacobson. „A formalization of darcs patch theory using inverse semigroups“. In: *Available fro m ftp://ftp. math. ucla. edu/pub/camreport/cam09-83. pdf* (2009).

[30] Jetbrains. *IntelliJ Platform SDK DevGuide*. "2019". URL: http://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi_elements.html.

[31] Joel Jones. „Abstract syntax tree implementation idioms“. In: *Proceedings of the 10th conference on pattern languages of programs (plop2003)*. 2003, pp. 1–10.

[32] Muris Lage Junior and Moacir Godinho Filho. „Variations of the kanban system: Literature review and classification“. In: *International Journal of Production Economics* 125.1 (2010), pp. 13–21.

[33] David Kawrykow and Martin P Robillard. „Non-essential changes in version histories“. In: *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE. 2011, pp. 351–360.

[34] Shuvendu K Lahiri et al. „Symdiff: A language-agnostic semantic diff tool for imperative programs“. In: *Computer Aided Verification*. Springer. 2012, pp. 712–717.

[35] *Language Server Protocol Specification*. Rev. 3.14.0. Microsoft. Dec. 2018. URL: https://microsoft.github.io/language-server-protocol/specification.

[36] André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. „Exception handling for error reporting in parsing expression grammars“. In: *Brazilian Symposium on Programming Languages*. Springer. 2013, pp. 1–15.

[37] Sérgio Medeiros and Fabio Mascarenhas. „Syntax error recovery in parsing expression grammars“. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. ACM. 2018, pp. 1195–1202.

[38] Sérgio Queiroz de Medeiros and Fabio Mascarenhas. „Error recovery in parsing expression grammars through labeled failures and its implementation based on a parsing machine“. In: *Journal of Visual Languages & Computing* 49 (2018), pp. 17–28.

[39] Samuel Mimram and Cinzia Di Giusto. „A Categorical Theory of Patches“. In: *Electronic Notes in Theoretical Computer Science (ENTCS)* 298 (2013), pp. 283–307.

[40] Eugene W Myers. „An O (ND) difference algorithm and its variations“. In: *Algorithmica* 1.1 (1986), pp. 251–266.

[41] Eugene W Myers and Webb Miller. „Optimal alignments in linear space“. In: *Bioinformatics* 4.1 (1988), pp. 11–17.

[42] Stas Negara et al. „Is it dangerous to use version control histories to study source code evolution?“ In: *ECOOP 2012–Object-Oriented Programming*. Springer, 2012, pp. 79–103.

[43] Martin Odersky et al. *The Scala language specification*. 2004.

[44] C Michael Pilato, Ben Collins-Sussman, and Brian W Fitzpatrick. *Version Control with Subversion: Next Generation Open Source Version Control*. " O'Reilly Media, Inc.", 2008.

[45] Romain Robbes. „Of change and software“. PhD thesis. Università della Svizzera italiana, 2008.

[46] Marc J Rochkind. „The source code control system“. In: *IEEE Transactions on Software Engineering* 4 (1975), pp. 364–370.

[47] David Roundy. „Darcs: distributed version management in haskell“. In: *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*. ACM. 2005, pp. 1–4.

[48] Miles Sabin. *Add support for multiple trailing implicit parameter list*. GitHub. URL: https://github.com/scala/scala/pull/5108.

[49] Jeff Sauro and James R Lewis. „Standardized usability questionnaires“. In: *Quantifying the user experience* (2012), pp. 185–240.

[50] Dennis Shasha and Kaizhong Zhang. „Fast algorithms for the unit cost editing distance between trees“. In: *Journal of algorithms* 11.4 (1990), pp. 581–621.

[51] Ivan Z. Siu. *git commit best practices*. Stack Exchange. eprint: http://stackoverflow.com/questions/6543913/. URL: http://stackoverflow.com/questions/6543913/.

[52] Codice Software. *semanticmerge*. Version 1.0.80.0. Jan. 27, 2016. URL: https://www.semanticmerge.com/.

[53] Diomidis Spinellis. „Version control systems“. In: *Software, IEEE* 22.5 (2005), pp. 108–109.

[54] Walter F Tichy. „Design, implementation, and evaluation of a revision control system“. In: *Proceedings of the 6th international conference on Software engineering*. IEEE Computer Society Press. 1982, pp. 58–67.

[55] R Hevner Von Alan et al. „Design science in information systems research“. In: *MIS quarterly* 28.1 (2004), pp. 75–105.

[56] Tim A Wagner. „Practical algorithms for incremental software development environments“. PhD thesis. Citeseer, 1997.

[57] Tim A Wagner and Susan L Graham. „Incremental analysis of real programming languages“. In: *ACM SIGPLAN Notices*. Vol. 32. 5. ACM. 1997, pp. 31–43.

[58] Peter Weissgerber and Stephan Diehl. „Identifying refactorings from source-code changes“. In: *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE. 2006, pp. 231–240.

[59] David S Wile. „Abstract syntax from concrete syntax“. In: *Proceedings of the 19th international conference on Software engineering*. ACM. 1997, pp. 472–480.

[60] Niklaus Wirth et al. *Compiler construction*. Vol. 1. Citeseer, 1996.

A Appendix

A.1 User Manual

A.1.1 Introduction

PolyCoDif is a tool for comparing different versions of source code stored in Git. Instead of the normal text-based change-set known from tools like *GitHub*, `git diff`, and *IntelliJ Idea*, this prototype tries to extract semantic information from the source code and presents a view of the change-set based on this semantic information. This means that *PolyCoDif* parses the code and generates the change-set based on the blocks of the code. Since it tracks continuous chunks of code which belong together, it is possible to depict when code was moved and not only if certain lines of code where added and removed.

The goal for PolyCoDif is to compliment classic diff tools and make it easier to review and understand the history of code.

A.1.2 Overview of the UI

The UI is split into four parts. The top contains the menu which shows how future extensions may look like – but does not serve any uses currently. The bottom fifth of the screen contains the history, a graphical time-line of the commits. With this view, one can select the time-range for which PolyCoDif should calculate the change-set. The first comparison point is selected by pressing the left mouse button. Dragging and releasing the mouse locks the second comparison point and updates the other screens. A screen-shot of this can be found in Figure A.1. To select for which file PolyCoDif should show the semantic difference, a tree browser on the left hand side can be used. Clicking on a file will automatically update the center of the screen and show the semantic change set. For this to work a time-range needs to be selected.

A.1.3 Change-Set UI

The change-set consists of a list of elements, each describing a semantic operation. There are four kinds of operations.

Add An *add* operation indicates that a new section was inserted into the program. Depending on the language there are several kinds of sections which might have been added. Examples for these are *classes*, *objects*, *methods* or *variables*.

Remove This indicates that a semantic code block was deleted from the program structure. The *remove* operation is the semantic dual from the *add* operation.

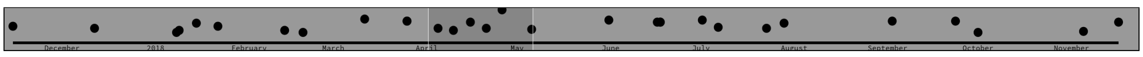
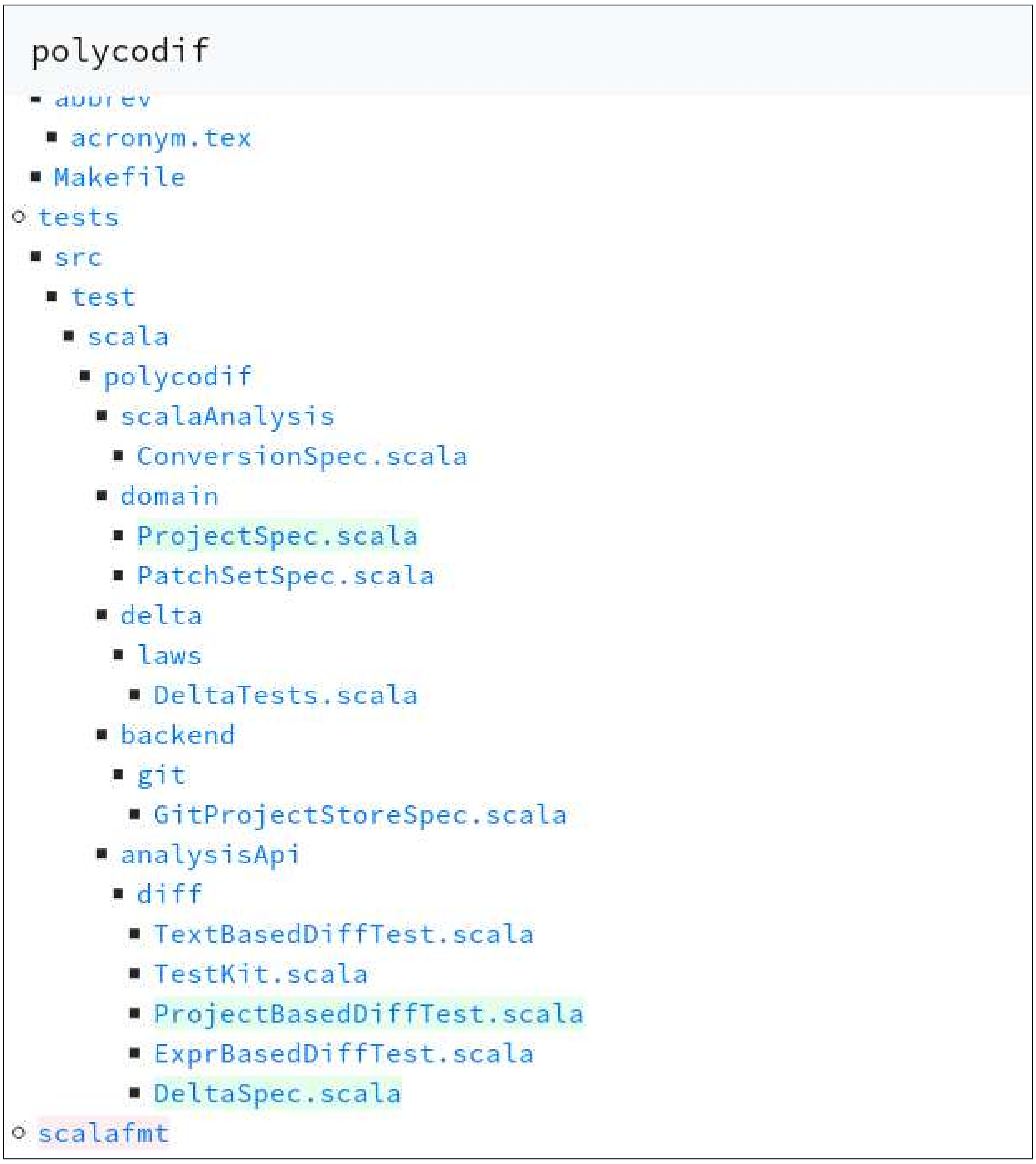


Figure A.1: Timeline



```
polycodif
  ▪ abbrev
    ▪ acronym.tex
  ▪ Makefile
○ tests
  ▪ src
    ▪ test
      ▪ scala
        ▪ polycodif
          ▪ scalaAnalysis
            ▪ ConversionSpec.scala
          ▪ domain
            ▪ ProjectSpec.scala
            ▪ PatchSetSpec.scala
          ▪ delta
            ▪ laws
              ▪ DeltaTests.scala
          ▪ backend
            ▪ git
              ▪ GitProjectStoreSpec.scala
          ▪ analysisApi
            ▪ diff
              ▪ TextBasedDiffTest.scala
              ▪ TestKit.scala
              ▪ ProjectBasedDiffTest.scala
              ▪ ExprBasedDiffTest.scala
              ▪ DeltaSpec.scala
○ scalafmt
```

Figure A.2: Outline

Move A *move* operation changes the position where a block is located in the program structure. There are no changes to the code it self, only the position of the code. Depending on the scope, this does not necessarily mean that the two versions of the program are semantically identical.

Update This operation combines the *move* operation with the standard text-based diff.

A.2 Questionnaire

A.2.1 Personal Questions

	Strongly Disagree				Strongly Agree
I have a lot of experience developing software	☐	☐	☐	☐	☐
I have a lot of experience reviewing code	☐	☐	☐	☐	☐
I have a lot of with **Scala**	☐	☐	☐	☐	☐
I have a lot of with **Java**	☐	☐	☐	☐	☐
I have a lot of with **Javascript**	☐	☐	☐	☐	☐

My preferred tool to write code is

☐ IntelliJ IDEA ☐ Eclipse ☐ Visual Studio ☐ VSCode ☐ other:

My preferred tool to review code is

☐ IntelliJ IDEA ☐ GitHub ☐ git command ☐ BitBucket ☐ other:

A.2.2 Scenario CRB

Link to Scenario http://polycodif.sperl.me:9002/ui#.

On a scale of 1 to 10, are there significant changes to the source code which effect the outcome of the program.

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9 ☐ 10

	Strongly Disagree				Strongly Agree
I found the system was useful to answer the question for this scenario.	☐	☐	☐	☐	☐
I would consider using the system in the future if I encounter a similar scenario.	☐	☐	☐	☐	☐

A.2.3 Scenario CRW

Link to Scenario http://polycodif.sperl.me:9003/ui#.

On a scale of 1 to 10, are there significant changes to the source code which effect the outcome of the program.

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9 ☐ 10

	Strongly Disagree				Strongly Agree
I found the system was useful to answer the question for this scenario.	☐	☐	☐	☐	☐
I would consider using the system in the future if I encounter a similar scenario.	☐	☐	☐	☐	☐

A.2.4 Scenario CCB

Link to Scenariohttp://polycodif.sperl.me:9000/ui#.

Are there any new methods introduced? ☐ Yes ☐ No
If yes, what are these methods called?

	Strongly Disagree				Strongly Agree
I found the system was useful to answer the question for this scenario.	☐	☐	☐	☐	☐
I would consider using the system in the future if I encounter a similar scenario.	☐	☐	☐	☐	☐

A.2.5 Scenario CCW

Link to Scenario http://polycodif.sperl.me:9001/ui#.

Are there any new methods introduced? ☐ Yes ☐ No
If yes, what are these methods called?

	Strongly Disagree				Strongly Agree
I found the system was useful to answer the question for this scenario.	☐	☐	☐	☐	☐
I would consider using the system in the future if I encounter a similar scenario.	☐	☐	☐	☐	☐

A.2.6 Overall Usability

	Strongly Disagree				Strongly Agree
I think that I would like to use this system frequently.	☐	☐	☐	☐	☐
I found the system unnecessarily complex.	☐	☐	☐	☐	☐
I thought the system was easy to use.	☐	☐	☐	☐	☐
I think that I would need the support of a technical person to be able to use this system.	☐	☐	☐	☐	☐
I found the various functions in this system were well integrated.	☐	☐	☐	☐	☐
I thought there was too much inconsistency in this system.	☐	☐	☐	☐	☐
I would imagine that most people would learn to use this system very quickly.	☐	☐	☐	☐	☐
I found the system very cumbersome to use.	☐	☐	☐	☐	☐
I felt very confident using the system.	☐	☐	☐	☐	☐
I needed to learn a lot of things before I could get going with this system.	☐	☐	☐	☐	☐