UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

IMPROVING SOURCE CODE CHANGE SET ANALYSIS BY USING A VISUAL TOOL

TESIS PARA OPTAR AL GRADO DE
MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN
MEMORIA PARA OPTAR AL TITULO DE INGENIERO CIVIL EN COMPUTACIÓN

RONIE ALEXIS SALGADO FAILA

PROFESOR GUÍA:
ALEXANDRE BERGEL

MIEMBROS DE LA COMISIÓN:
JOCELYN SIMMONDS WAGEMANN
ÉRIC TANTER
LUCIANO AHUMADA FIERRO

SANTIAGO DE CHILE
2019

Resumen

En los procesos modernos de desarrollo de programas computaciones, los elementos nuevos del mismo son desarrollados en ramas completamente aisladas por cada uno de los miembros de un equipo de desarrollo. La integración de cada una de estas ramas debe ser aprobada por un integrador, quien debe revisar manualmente los cambios introducidos al código fuente dentro de estas ramas. Esta revisión es un proceso largo y tedioso, que normalmente se realiza mediante la lectura detallada de las diferencia textuales en los cambios al código fuente.

La dificultad de realizar esta revisión tiene un impacto negativo en la aceptación de estos cambios. Ha sido mostrado que esto puede implicar el rechazo a importantes correcciones de errores al código de un programa computacional.

Con el objetivo de facilitar la tarea de revisión de cambios de código, se diseño e implemento una herramienta visual. Esta herramienta ayuda en el análisis de cambios de código fuente al proveer diferentes vistas sobre los cambios: una vista general con métricas sobre los cambios, y una visualización representando los cambios estructurales en el código fuente. Esta herramienta visual fue evaluada mediante la realización de un experimento controlado con desarrolladores.

Nuestros resultados experimentales no muestran diferencias significativas entre utilizar la herramienta visual, y una herramienta de diferencia textual en los siguientes aspectos: tiempo de análisis, precisión del análisis e inferencia de la intención de los cambios. Sin embargo, si se encontró una percepción positiva en la experiencia de usuario, y una carga mental reducida al utilizar la herramienta visual.

Abstract

Modern software development is performed by developing features in isolated branches by each member of a software development team. When these branches need to be integrated, they have to be manually reviewed by an integrator. Source code reviewing can be a tedious and time consuming task, which is normally performed by hand reviewing a textual diff of the change set.

The difficulty of reviewing source code changes can have a negative impact on the acceptance of these changes. It has been shown that this can cause the rejection of important bug fixes for a software project.

In order to facilitate the task of reviewing source code change sets, we designed and implemented a visual tool. Our visual tool helps in assessing source code change sets by providing different views of the change set: an overall overview of the change set with metrics, and a visualization representing the structural changes in the source code. We evaluated our visual tool by performing a controlled experiment with software practitioners.

Our experimental results show no significant differences between using our visual tool and a textual diff tool in the following terms: time of analysis, precision of the analysis, and inference of intention of changes. However, we did find a positive user perception, and a reduced mental load when using our visual tool.

I am dedicating this thesis to my mother. Rest in peace.

Acknowledgements

First, I like to thank my thesis advisor, Prof. Alexandre Bergel. I first met Alexandre in 2013 during a software engineering lecture where he noticed my programming skills and offered me a job doing graphics in Pharo. That is how I entered into research, and thanks to all of his support of my work I achieved many interesting things.

Second, I like to thank the people from Inria, Lille, for receiving me several times in their lab, including the five-months internship that I did there. I specially thank to Prof. Stephane Ducasse and Dr. Marcus Denker for receiving me in their research team.

Third, I would like to thank the people from Inria, Chile, for participating in my controlled experiments, and allowing me to use their office some meetings with my advisor.

I thank Vanessa Peña, for helping me improve the design of the Git Thermite user experience, and helping me on the design and the analysis of the controlled experiments.

I thank the people of my job in AOne Games SpA. I thank my boss and work colleagues for allowing to keep working on this thesis along the job. I thank them for allowing me to conduct some of the controlled experiments in the company office.

I would like to thank Mr. Renato Cerro for helping me on doing English corrections for this thesis.

And the most important, I would like to thank my family. I thank them for raising me and supporting me for all of this years.

Contents

1 Introduction **1**
- 1.1 Problem description . . . 1
- 1.2 Problem statement . . . 3
- 1.3 Hypothesis . . . 4
- 1.4 Objectives . . . 4
- 1.5 State of the art . . . 4
- 1.6 Methodology . . . 4
- 1.7 Git Thermite . . . 5
- 1.8 Git Thermite design evolution . . . 6
- 1.9 Controlled experiments . . . 6
- 1.10 Controlled experiment results . . . 7
- 1.11 Extra informal supporting data . . . 7
- 1.12 Discussion and conclusions . . . 7

2 State of the art **8**
- 2.1 Textual diff tools . . . 8
  - 2.1.1 Unix *diff* tool . . . 8
  - 2.1.2 GitHub diff . . . 9
  - 2.1.3 Monticello diff . . . 10
- 2.2 GitHub pull requests . . . 11
- 2.3 Source code metamodeling . . . 13
- 2.4 Software structural visualizations . . . 14
  - 2.4.1 Polymetrics views . . . 14
  - 2.4.2 CodeCity . . . 15
  - 2.4.3 CuboidMatrix . . . 16
- 2.5 Software change set visualizations . . . 18
  - 2.5.1 Torch . . . 18
  - 2.5.2 Motive . . . 21
- 2.6 GitHub visualizations . . . 22

3 Methodology **24**
- 3.1 Tool design . . . 25
- 3.2 Controlled experiment design . . . 25
- 3.3 Pilot experiment . . . 26
- 3.4 Pilot experiment analysis . . . 26
- 3.5 Tool design refinement . . . 27

3.6 Final controlled experiment execution . . . 27
3.7 Final controlled experiment analysis . . . 28
3.8 Gathering of extra informal data . . . 28

4 Git Thermite **30**
4.1 Getting started with the tool . . . 30
4.1.1 Installing Git Thermite . . . 30
4.1.2 Visualizing pull requests . . . 32
4.1.3 Visualizing local git commits . . . 32
4.1.4 Monticello integration . . . 33
4.1.5 Iceberg integration . . . 34
4.2 Color Legend . . . 35
4.3 Unchanged source code elements and cluttering . . . 36
4.4 Business Card Visualization . . . 37
4.5 File changes visualization . . . 39
4.6 Structural changes visualizations . . . 40
4.7 Class changes visualization . . . 42
4.8 Method/function changes visualization . . . 43
4.9 Global Textual diff . . . 44

5 Git Thermite design evolution **45**
5.1 First version: an attempt to make something original . . . 45
5.2 Second version: adding elements from Torch . . . 49
5.3 Third version: multiples visualizations . . . 51
5.4 Final version: consolidating the tool design . . . 53

6 Controlled experiment design **55**
6.1 Similar controlled experiments . . . 56
6.2 Research questions and hypotheses . . . 57
6.3 Controlled experiment tasks and questions . . . 57

7 Pharo controlled experiment **61**
7.1 Questions . . . 61
7.1.1 Personal questions . . . 61
7.1.2 Source code change set questions . . . 62
7.1.3 Questions about the user general impressions on the tool . . . 63
7.2 Pilot . . . 63
7.3 Execution . . . 64
7.4 Results . . . 68
7.4.1 Unfamiliar code analysis . . . 69
7.4.2 Familiar code analysis . . . 76
7.4.3 General impressions . . . 81
7.4.4 NASA TLX form . . . 82

8 Informal Evaluation **84**
8.1 Asking an open source community . . . 84
8.2 Feedback from an open source community . . . 85

9 Discussion **87**
9.1 Hypothesis testing . . . 87
9.1.1 Differences in change set analysis time . . . 87
9.1.2 Change set analysis precision . . . 89
9.1.3 Intention of changes inference . . . 90
9.1.4 Cognitive load . . . 90
9.2 Threats to validity . . . 91
9.3 Limitations in our implementation . . . 92

10 Conclusions **94**

Bibliography **97**

A Python controlled experiment attempt **100**
A.1 Questions . . . 100
A.1.1 Personal questions . . . 100
A.1.2 Source code change set questions . . . 101
A.1.3 Questions about the user general impressions on the tool . . . 101
A.2 Pilot . . . 102
A.3 Execution . . . 102
A.4 Experimentation problems . . . 104
A.5 Results . . . 105
A.5.1 Unfamiliar code analysis . . . 105
A.5.2 Familiar code analysis . . . 107
A.5.3 General impressions . . . 109
A.5.4 NASA TLX form . . . 109

B Git Thermite learning material **111**

C Controlled experiment exact answer sample **121**

List of Tables

6.1 Null and alternate hypotheses. 58

7.1 Pharo experiment change sets selected from Roassal2. 64
7.2 Pharo experiment participants background. 66
7.3 The experience of Pharo participants with visualizations, Git and GitHub. . 67
7.4 Pharo participants experience with source code reviewing. 68
7.5 Pharo experiment change set assignment 68
7.6 Unfamiliar Pharo code: grading of question *Characterize the size and complexity of the commits (trivial vs complicated)* 70
7.7 Unfamiliar Pharo code: description of changes. 72
7.8 Unfamiliar Pharo code: presence of refactoring. 73
7.9 Unfamiliar Pharo code: class hierarchy description. 75
7.10 Unfamiliar Pharo code: difficulty of understanding changes (1 difficult, 5 easiest), and time taken by the participants to analyze a change set. 76
7.11 Familiar Pharo code: grading of question *Characterize the size and complexity of the commits (trivial vs complicated)* 77
7.12 Familiar Pharo code: description of changes. 78
7.13 Familiar Pharo code: presence of refactoring. 79
7.14 Familiar Pharo code: class hierarchy description. 80
7.15 Familiar Pharo code: difficulty of understanding changes, and time of analysis. 81
7.16 Pharo experiment general impression agreement questions. These questions are in a scale between 1 (strongly disagree) and 5 (strongly agree) inclusive. 82
7.17 NASA TLX responses with Git Thermite in the Pharo experiment. Scale from 1 to 21, inclusive. Middle point at 11. 83
7.18 NASA TLX responses with the *Monticello textual* diff tool in the Pharo experiment. Scale from 1 to 21, inclusive. Middle point at 11. 83
7.19 NASA TLX Diff responses minus Git Thermite responses. p-value is computed by using a pairwise two-tailed t-Student test. 83

9.1 Unfamiliar Pharo code: experimentation time analysis. Times are in minutes. 88
9.2 Familiar Pharo code: experimentation time analysis. Times are in minutes. . 88

A.1 Selected change sets for the Python experiment. 102
A.2 Python experiment participants background. 103
A.3 Python participant experience with visualizations 103
A.4 Python participants experience with Git and GitHub. 103
A.5 Textual diff tool used by the Python practitioners 103

A.6 Python participants experience with source code reviewing. 104
A.7 Python experiment change set assignment 104
A.8 Unfamiliar Python code: size and complexity. 105
A.9 Unfamiliar Python code: Presence of tests 106
A.10 Unfamiliar Python code: description of changes. 106
A.11 Unfamiliar Python code: difficulty of understanding changes, and time of analysis. 107
A.12 Familiar Python code: size and complexity. 107
A.13 Familiar Python code: description of changes. 108
A.14 Familiar Python code: difficulty of understanding changes, and time of analysis. 108
A.15 Python experiment general impression agreement questions. These questions are on a scale of 1 (strongly disagree) and 5 (strongly agree) inclusive. 109
A.16 NASA TLX responses with Git Thermite in the Python experiment. Scale from 1 to 21, inclusive. Middle point at 11. 109
A.17 NASA TLX responses with the *GitHub diff* tool in the Python experiment. Scale from 1 to 21, inclusive. Middle point at 11. 110
A.18 NASA TLX Diff responses minus Git Thermite responses. p-value is computed by using a pairwise two-tailed t-Student test. 110

List of Figures

1.1 Git Thermite visualization for *keras-team/keras* pull request number 10130. . 5

2.1 GitHub diff for a file in pull request. 10
2.2 Pharo class browser displaying the source code for a method. 10
2.3 Monticello diff between PetitParser commit number 289, and commit number 290. 11
2.4 Pull request creation form on GitHub. 12
2.5 Discussion thread for a pull request. This is pull request number 411 or PR number 411 of the OpenSmalltalk/opensmalltalk-vm project on GitHub. This pull request has two discussion comments. In the lower section, there are links and checkboxes with the results from two different continuous integration servers. 12
2.6 Buttons at the end of the GitHub pull request interface. 13
2.7 Example of a polymetric view (Image from [1]). Part (a) shows how a single node can represent up to five different metrics [1]. Part (b) shows a *system complexity view* with nodes representing a class hierarchy on a tree layout [1]. The metrics used in this visualization are: the number of class attributes on the width, the number of methods on the height, and the color shade represents the number of lines of code per class [1]. 14
2.8 CodeCity software visualization (Image from [2]). This visualization maps the number of methods of classes to the *height* of the building. The *base size* (width and depth) is mapped to the number of lines of code of each class. The color of these classes is used to display *design problems* such as: brain class (yellow), god class (red), a class that is both a brain and god class simultaneously (orange), data class (purple). This particular CodeCity software visualization is called a *disharmony map* [2] because of the presence of this last metric (design problems). 15
2.9 CuboidMatrix visualization (Image from [3]). This visualization represents the interaction between classes along the execution of a program. The Z axis represents time, and the X and Y axis are used for labeling classes. The presence of a cube indicates that a pair of classes (X and Y axes) interacted during a given time frame (Z axis). The size of a cube represents the number of interactions, and a color gradient from blue to yellow is used to represent the duration of these interactions. 17
2.10 The Torch dashboard (Image from [4, 5]). 18
2.11 The Torch structural and condensed changes visualizations (Image from [5]). 19
2.12 The Torch symbolic cloud visualization (Image from [5]). 20

2.13 Motive software visualization (Image from [6]). 21
2.14 GitHub contribution matrix. 22
2.15 GitHub fork graph 22
2.16 GitHub chart of contributions to a repository. 23

3.1 Methodology flow diagram 24

4.1 Pharo image desktop. Part A is the *World menu*. Part B is the *Playground* opened with the world menu. Part C is the *Do it* button for executing the Pharo script in the *Playground*. Part D is the *Git Thermite* menu that is added after installing Git Thermite on the Pharo image. 31
4.2 The *World menu* entries that are added by the Git Thermite installation. . . 32
4.3 Dialog for starting a pull-request visualization. 32
4.4 Dialog for starting a local commit visualization. 33
4.5 Git Thermite integration on the Monticello browser. The *Thermite changes* button allows comparing the selected commit with the working copy in Pharo image. The *Thermite parent changes* button allows comparing the selected commit with an arbitrary ancestor commit. This ancestor commit is selected from drop-down list that appears after clicking on the *Thermite parent changes* buttons. 33
4.6 Git Thermite integration on the Iceberg repository browser. 34
4.7 Git Thermite integration on the Iceberg repository browser. 35
4.8 Color legend 35
4.9 Visualization with unchanged lines, unchanged class and unchanged methods. This is *numpy/numpy* pull request number 8939. 36
4.10 Business card visualization for *numpy/numpy* pull request number 8939. This visualization corresponds to the same pull request that is visualized without filtering on Figure 4.9. 37
4.11 File changes visualization accessing button 39
4.12 File changes visualization for *numpy/numpy* pull request number 8939. This corresponds to the same pull request that is visualized in Figure 4.10. 39
4.13 Code changes visualization accessing button 40
4.14 Code changes visualization for *pharo-project/pharo* pull request number 166. 40
4.15 Method description tool-tip on the left, with similar methods highlighted. This is in *pharo-project/pharo* pull request number 166. 42
4.16 The textual diff of a Python method in *numpy/numpy* pull request number 8939. 42
4.17 Classes accessing button 42
4.18 Class changes visualization for *numpy/numpy* pull request number 8939. . . 43
4.19 Methods accessing button 43
4.20 Method visualization for *numpy/numpy* pull request number 8939. 43
4.21 Global textual diff accessing button 44
4.22 Global diff visualization 44

5.1 Business card visualization of the first iteration in the design of Git Thermite for *numpy/numpy* pull request number 9285. Part 1 is the title bar, part 2 is a *polymetric view* with the modified files, and part 3 is another *polymetric view* with the source code structure. 46

5.2 Highlighting an element in a visualization with the cursor displays one of these tooltips. 47
5.3 Commit tree visualization for *numpy/numpy* pull request number 8939 (Only first three levels). The vertical axis represents time, and each one of the boxes represents a commit in the history of the project. This visualization is an attempt to display the history of the change sets, however in its current state it does not provide helpful information. 48
5.4 Second tool design iteration business card visualization for *numpy/numpy* pull request number 8939. 49
5.5 New version of the commit tree visualization *numpy/numpy* pull request number 8939 (Only first three levels). This version uses the new way for representing added and removed lines. Compare this version of the visualization with the previous one in Figure 5.3 50
5.6 Third tool design iteration business card visualization for *numpy/numpy* pull request number 8939. Part A is the title bar of the business card. Part B is the section with metrics. Part C is the section with navigation buttons. Notice that the colored bar with the number of lines of code is dominated by the gray colored bar representing unmodified lines which makes it difficult differentiate the size between green (added lines) and red colored bars (removed lines). . . 51
5.7 Third tool design iteration code changes visualization for *pharo-project/pharo* pull request number 166. 52
5.8 Final tool design iteration business card visualization for *numpy/numpy* pull request number 9285. 53

7.1 Business card visualizations for the different change sets. 65
7.2 Structural changes for CS1 and CS2. 66
7.3 Structural changes for CS3 and CS4. 67

Chapter 1

Introduction

1.1 Problem description

Software development is usually performed with the assistance of a version control system (VCS) whose job is to record the full history of a software development process. The recording of this history is done typically with one or more of the following objectives:

- Reverting changes that introduce regressions [7].
- Detecting a change that introduces a regression by performing a binary search on the project history.[1]
- Integrate the collaborative work by a team composed with multiple people [8].
- Having multiple development branches for having:

  A mainstream branch that has been already subjected to a source code reviewing and approval process.

  Different source code branches with changes that still need to be subjected to code reviewing, and be approved by a quality assurance (QA) team [8] for integration into the mainstream branch.

These points typically apply to any kind of software project, including open source software projects and closed source proprietary software. The difference between the nature of an open source software, or a proprietary closed source software affect in which of these points can be more significant to the software development process. For example, in an open source project it is typical for a project to receive contributions from developers that do not belong to the core development team (*i.e.,* the original authors). This means that an open source project can impose a large burden to the integrator of the project, to perform a detailed source code review of the contributions. As a counter example, in the case of a small proprietary software company, the whole development team sits in the same building, or in the same office, so it is easier for the developers who work on a bug fix or feature to communicate with the integrator and the company QA team.

[1] *git bisect* is a tool for automating this process: `https://git-scm.com/docs/git-bisect`

Comparing changes. Despite the fact that the focus of using a version control system can be in avoiding serious business problems (*e.g.,* a critical bug introduced before a release), or in simplifying the integration of changes, there is still the necessity to manually review of code changes. Most of the different version control systems such as Perforce[2], SVN[3], Monticello[4], or Git[5] only provide a textual diff interface as a way to compare the changes introduced by a commit, or the changes between two different versions. The textual diff interface of these different tools typically presents the code changes in one of the following ways:

- The Unix *diff* tool result: added lines with a plus (+) symbol in the beginning, optionally colored in green; removed lines with a minus (−) in the beginning, optionally colored in red; and some surrounding lines to give a bit of context.
- A graphical interactive text editor with two panels: One panel with the old version, and another panel with the new version. Boxes indicating which paragraphs of text are added, removed, moved or remain unchanged.

Git and pull requests. In the particular case of the Git version control system, there are some collaborative social platforms such as GitHub[6], and GitLab[7] that augment Git features by providing facilities for proposing a *pull request*. Since Git is a distributed version control system, when a developer works with a Git repository on a project, the developer works with a full clone of the mainstream Git repository. A clone of a Git repository keeps a copy of the whole history of the project. This design of Git enables developers to work independently from one or another, until they want to push their changes into the mainstream repository of the software. If the developers have the needed permissions, they can just push their changes into the mainstream. However, typically only the integrator has the permission for pushing code into the mainstream repository. To avoid giving commit permissions to everyone, the typical flow with Git for introducing changes into the mainstream is to submit a *pull request* to the integrator. A pull request is a formal request to an integrator to accept some Git commits into its repository. When an integrator receives a pull request, he can review the changes before deciding to accept or reject the pull request.

In the case of GitHub pull requests, each pull request has an associated discussion thread where the users post comments and discuss the pull request. If the pull request does not introduce conflicts, in the case of GitHub they can be integrated by just clicking on a button. Otherwise, conflicts must be resolved manually by the integrator, or by the user who submits the pull request. When the pull request submitter pushes more changes into the branch that he used for creating the pull request, these changes are added automatically into the pull request by GitHub. By committing more changes in this way, the submitter can comply with extra requirements imposed in the pull request discussion thread, or even resolve merge conflicts.

[2]Perforce: http://www.perforce.com/
[3]Apache Subversion https://subversion.apache.org/
[4]Monticello: http://pharobooks.gforge.inria.fr/PharoByExampleTwo-Eng/latest/Monticello.pdf
[5]Git: https://git-scm.com/
[6]GitHub: https://github.com/
[7]GitLab: https://about.gitlab.com/

Pull request acceptance factors. The acceptance factors for an open source pull request can be divided into two categories: social, and technical [9, 10]. Some of these acceptance factors can be described by answering one of the following questions about the pull request:

- Is the open source project mature, conservative, and tending to stability? (Social) [9]
- Are the pull requests made by a developer making bug fixes, adding new features or both? (Technical)
- Do the pull requests include tests? (Technical) [9]
- How many other open source projects depend on this open source project (*i.e.,* Audience Pressure)? (Social) [9]
- How many discussion comments has the pull request made by the developer? (Social and technical) [9]
- Does the developer have prior contributions to this open source project? (Social) [9, 10]
- Which programming languages does the developer tend to use? (Technical) [10]
- What are the sizes of the commits made by the developer? (Social and technical) [9]
- How many files does the developer tend to modify, add or remove in his pull requests? (Technical) [9, 10]
- Is the developer a contributor or a main collaborator of the project? (Social) [9]

These acceptance factors are important because they show a preponderance of social factors above technical factors, especially for developers that are outsiders or are newcomers in an open source community. These factors are also important because they lend an insight on what technical and source code metrics are used for deciding on whether to accept or reject a contribution. Most of these technical factors are not evident, or are hard to deduce by just reading the change set textual diff. These analyses are especially hard for very large change sets.

1.2 Problem statement

From the previous description, we state the following research problem:

How do we facilitate the reviewing and comprehension of code changes during the evolution of a software?

This problem is highly significant for open source and proprietary software communities because doing manual source code revision can be a tedious and time consuming task. In the case of the open source community, this can be one of the reasons for the tendency to reject large pull requests. The acceptance of contributions to open source projects is essential, because these contributions contain important bug fixes, and new features that are used by the community. It may be possible to reduce the influence of social acceptance factors by facilitating the reviewing process of the contributions to open source project. However, proving this causal relationship between the difficulty of the reviewing process and the contributions to open source projects, it is beyond the scope of this thesis.

1.3 Hypothesis

The research hypothesis for this thesis is the following:

By using a visual analysis and reporting tool that displays metrics, the changes to the structure of the software and the source code, in addition to the information presented by the textual diff, it is possible to facilitate code reviewing process in terms of time, comprehension of the changes and cognitive load[8] *to the reviewers.*

1.4 Objectives

The main objective objective of this thesis is to test this hypothesis by building a visual tool for assessing software changes, and then validating it with professional practitioners. Due to the broadness of this main objective, this thesis focuses on only achieving the following specific objectives as an approximation for the main objective:

- Define or use existing visual metaphors to precisely convey information about changes.
- Integrate metrics and visualizations about changes in a single tool, in order to enable a workflow for assessing the different aspects of a software change.
- Validate the tool with people for testing the hypothesis.

1.5 State of the art

Chapter 2 describes the relevant state of the art. Section 2.1 describes commonly used *textual diff tools*, which are the tools most commonly used for comparing source code change sets. Section 2.2 describes in detail the concepts and flow related with a *GitHub pull request*. The concept of source code meta modeling which is crucial for analyzing source code is described in Section 2.3. Generic software structural visualizations are described in Section 2.4. Software change set visualizations are described Section 2.5. Finally some other commonly used visualizations available in GitHub are described in Section 2.6.

Of particular importance is the work of Torch [4, 5] by Gomez *et al.*, which is used as the main source of inspiration for the design of Git Thermite. Torch is described in Section 2.5.1.

1.6 Methodology

The methodology followed by this thesis is an iterative design and evaluation process. This iterative process is summarized in the following steps:

[8] By cognitive load, we mean the mental stress or demand for performing a given task.

1. Design and build a tool that integrates metrics and visualizations about a software change set.
2. Design a controlled experiment with the designed visual tool, and a baseline tool.
3. Perform a pilot experiment with people with at least a basic knowledge of open source software developement.
4. Analyze and use the feedback from the pilot experiment. If there are critical design and usability flaws in the tool detected by testing with people, improve the design by going back to the first step.
5. Perform a final controlled experiment with professional practitioners.
6. Analyze the result of the final controlled experiment.
7. Gather informal feedback by testing the tool with the community, in order to reinforce the results obtained from the final controlled experiment.

This iterative design and testing loop was followed multiple times during the execution of this thesis. The final designed tool is called Git Thermite, and the final controlled experiment was conducted with five Pharo[9] practitioners.

A detailed description of the complete methodology is presented in Chapter 3.

1.7 Git Thermite

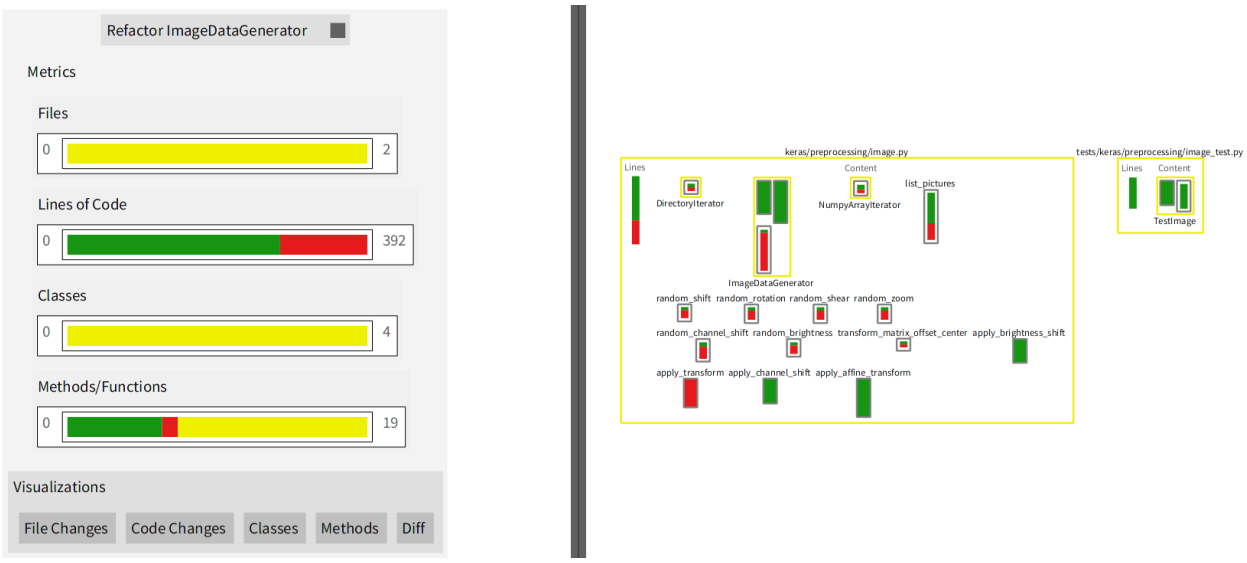


Figure 1.1: Git Thermite visualization for *keras-team/keras* pull request number 10130.

The visual tool for assessing source code change sets designed for this thesis is called Git Thermite. For an example of a visualization produced by this tool, see Figure 1.1. A detailed description of Git Thermite is presented in Chapter 4.

[9]Pharo is a dialect of the Smalltalk programming language. http://www.pharo.org/

1.8 Git Thermite design evolution

The design process for constructing the Git Thermite visual tool was an iterative process that produced four different versions of the tool. This evolution process is described in Chapter 5. Our original plan involved only a single designed tool through brainstorming sessions, and by adapting existent structural software visualizations for visualizing change sets. We initially tried to design something different than Torch based on the concepts of *polymetric visualizations* (See Section 2.4.1). When we tested this first version of the tool (See Section 5.1) in a pilot experiment with two Python practitioners from the industry, we immediately detected a serious design error in this first version of the visual tool: displaying the changed number of lines of code changes by using width and height is non-intuitive. This design error is evident and would strongly bias our results against the visual tool because of bad tool design. Since our objective is in producing a tool that is better than the commonly used textual diff tools, we decided to go back to the design phase.

User experience and user interface design is a difficult process [11]. Many design errors are only detected when trying a design with a person. User experience design is commonly carried on a participative framework that involves the end user [12, 13]. We learned this during our first tool design attempt (See Section 5.1), and our initially planned tool design process ended becoming a trial by error design process. This resulted in four iterations on our visual tool design (See Chapter 5). Each one of these iterations was tested with at least one pilot experiment with one or two participants. The participants of these pilot experiments are not carried to experiments of subsequent iterations, and they are not carried to the final controlled experiment.

1.9 Controlled experiments

For evaluating Git Thermite, we decided to conduct a controlled experiment. The overall design process and the guidelines chosen controlled experiment protocol are described in chapter Chapter 6.

We conducted this controlled experiment with Pharo practitioners. The exact questions asked during this controlled experiment are present in Section 7.1. The results of the individual responses along with a treatment to facilitate their comparisons are exposed in Chapter 7. In order to separate this experimental data from the analysis and the conclusions, Chapter 7 only does a minimal analysis of these responses. This data is further discussed and analyzed in Chapter 9.

We attempted to conduct another controlled experiment with Python practitioners which is described in Appendix A. Unfortunately, we encountered serious problems when conducting the Python experiment with our Python parser, so we did not use this data for the final results and conclusions of this thesis. However, this data still seems to display the same qualitative tendencies as the Pharo controlled experiment, and we are still including the data gathered during this experiment in Appendix A.

1.10 Controlled experiment results

A preliminary qualitative analysis of the data gathered in the Pharo controlled experiment show mostly negative results not favoring our visual tool in terms of: time of analysis taken by the participant; the precision of the analysis made by the participants; and the capabilities of the participant on inferring the intention of changes made by another developer in a commit. However, the written feedback given by the participants display a positive perception favoring the visual tool.

1.11 Extra informal supporting data

Since we got these negative results with our controlled experiment, we decided to gather extra informal data to additionally support the positive user perception hypothesis. This extra informal data is gathered by asking the members of an open source community to try the visual tool, and then to fill an online questionnaire. This gathering process and the obtained data is described in Chapter 8.

1.12 Discussion and conclusions

By analyzing the controlled experiment data we concluded the following results:

- There were no qualitative differences with the baseline tool in terms of time analysis taken by the participant. (See Section 9.1.1)
- There were no qualitative differences with the baseline tool in terms analysis precision. (See Section 9.1.2).
- There were no qualitative differences with the baseline tool in terms of inferring the intention of the changes. (See Section 9.1.3).
- There was a positive user perception of favoring the visual tool against the baseline tool. (See Section 9.1.4).

Chapter 2

State of the art

This chapter describes the relevant state of the art for this thesis. The first section of this chapter describes some existing textual diff tools that are commonly used in the software development industry (See Section 2.1). The second section of this chapter describes the process that involves a GitHub pull request, which is a popular way for submitting contributions to public open source projects (See Section 2.2). The third section of this chapter describes the concept of source code meta modeling, which is a crucial tool used for analyzing the structure of source code project before constructing a software visualization (See Section 2.3). The remaining sections of this chapter are focused on describing existing visualizations that are used to visualize the structure of a software in general (Section 2.4), visualizations for the changes in a software (See Section 2.5), and some other relevant visualizations that are included in GitHub (See Section 2.6).

2.1 Textual diff tools

2.1.1 Unix *diff* tool

The Unix command line *diff* program [14] is the traditional tool for comparing changes between text files. Diff is a command line tool that receives the path of one *old version* of the text file, and the path with a *new version* of the same text file. The result produced by *diff* is a *patch*, which is another text file that describes the changes needed to convert the *old version* of the text file into the *new version* of the text file. This *patch* file is composed of the lines of text that must be added with a plus (+) sign on the beginning of the line. The lines that must be removed are indicated with a minus (−) sign. The modified lines in a *diff patch* are surrounded by unchanged lines that show the surrounding modification context. The number of the line at the beginning of one of these patching context is specified by a line that starts with @@. The output of the *diff* tool is optionally colorized by assigning green to added lines, red to removed lines, and blue to the @@ control lines.

For an example of the result of the Unix textual diff tool, we include the source code of

Listing 2.1: Example source code listing for old-code.py

```
import sys

print 'Hello World, from', sys.argv[0]
for arg in sys.argv[1:]:
        print 'Hello ', arg
```

Listing 2.2: Example source code listing for new-code.py

```
import sys

print 'Good bye World, from', sys.argv[0]
for arg in sys.argv[1:]:
        print 'Good bye ', arg
```

a very simplistic Python program in two version: Listing 2.1 contains the old version of this Python program, and Listing 2.2 contains the new version of this same Python program. The result of running the *diff* with colored output is present in Listing 2.3.

Listing 2.3: Result of running *diff -u –color old-code.py new-code.py*

```
--- old-code.py 2019-05-24 22:07:17.398678873 -0400
+++ new-code.py 2019-05-24 22:07:37.991827581 -0400
@@ -1,6 +1,6 @@
 import sys

-print 'Hello World, from', sys.argv[0]
+print 'Good bye World, from', sys.argv[0]
 for arg in sys.argv[1:]:
-    print 'Hello ', arg
+    print 'Good bye ', arg
```

2.1.2 GitHub diff

GitHub diff is a web interface above the Unix *diff* tool (See Section 2.1.1) that is integrated on top of GitHub [15, 16]. This web interface is tightly integrated onto GitHub in two parts: the history on a GitHub repository, and the changes description for a pull request. The integration on GitHub makes this web interface one of the most commonly used version of the textual diff tool by open source project integrators, and individual contributors. Figure 2.1 displays an example diff for a GitHub pull request. This web interface uses the same color convention as the standard Unix textual diff tool: green for added lines, red for removed lines, and blue for contextual information (*e.g.,* line number).

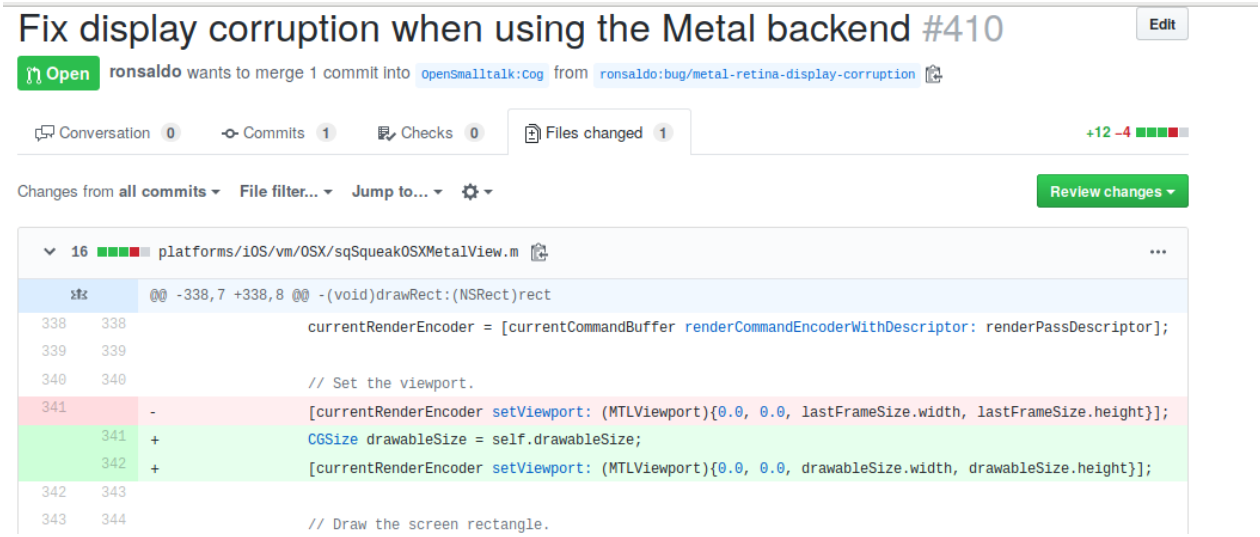


Figure 2.1: GitHub diff for a file in pull request.

2.1.3 Monticello diff

Monticello is the traditional version control system used by Smalltalk programming environment, and by Pharo in particular [17, 18]. Smalltalk, unlike other programming languages does not use *text files* for representing its source code. Instead, the Smalltalk programming environment are constructed around the notion of manipulating *live objects* in an environment. These objects are persisted into a single monolithic *image*. A Smalltalk *image* is a memory dump of the whole Smalltalk system, which includes:

- The Smalltalk compiler.
- The Smalltalk debugger.
- The Smalltalk browser for manipulating individual classes.
- Application-specific classes and methods.

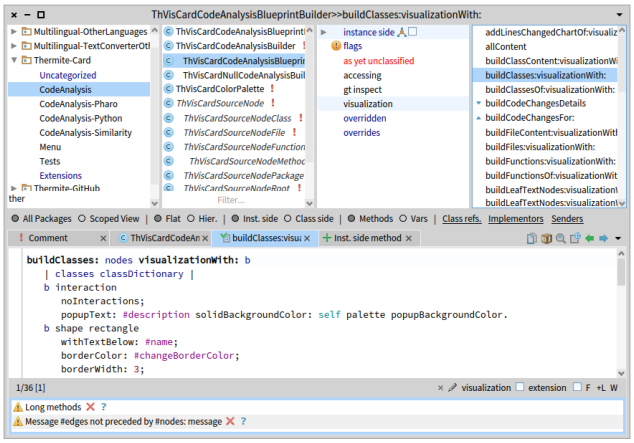


Figure 2.2: Pharo class browser displaying the source code for a method.

In other words, a Smalltalk image can also be described as a whole copy of a disk drive holding an operating system, applications and data. Smalltalk classes and methods are

constructed and manipulated by using a browser (See Figure 2.2). The source code for individual methods can only be modified by using a class browser. The source code for a single method is completely isolated from the source of other methods. The fact that the source code for Smalltalk systems is object oriented instead of text file oriented has the direct consequence that traditional text file based version control systems cannot be used directly for versioning Smalltalk source code. This also implies that the traditional text file based Unix diff tool (Section 2.1.1) is not be adequate for Smalltalk source code. For these reasons, Smalltalk systems have their own version control systems. Many Smalltalk dialects including Squeak and Pharo, use Monticello as their native version control system.

Figure 2.3: Monticello diff between PetitParser commit number 289, and commit number 290.

Monticello includes its own diff tool for comparing changes between specific version of Smalltalk code. For example, the diff in Figure 2.3 contains three columns:

1. List with changed classes and methods. An icon of the left side of a method indicates on whether the method is added (a green plus + sign), removed (a red × sign), or modified (a <> sign).
2. A column with an old version of the method in the middle.
3. A column with a new version of the method on the right.

For the reason of increasing open source project exposure, Pharo is currently migrating to use Git and GitHub (Since Pharo 7). In those environments, Pharo is serializing its source code in text files with a special format. These text files are not in the Smalltalk programming language, but they are isomorphic to Smalltalk classes and methods. Since these are normal text files, they can be used with the traditional text file based tools such as the Unix diff tool. This isomorphism between Smalltalk classes and methods also allows one to keep using the Monticello diff tool with Smalltalk projects on Git and GitHub.

2.2 GitHub pull requests

GitHub pull requests [19] are a collaborative tool for submitting and accepting contributions to an open source project. The typical life cycle of a pull request includes the following steps:

1. A contributor makes his or her changes in a separate Git branch on a private fork[1].

[1]A project fork is a copy of a mainstream project with divergent changes by an individual or group making

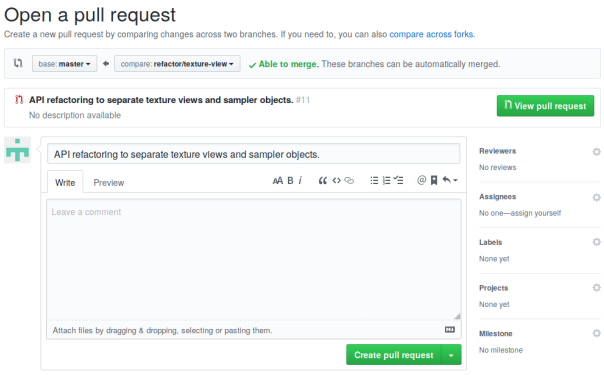


Figure 2.4: Pull request creation form on GitHub.

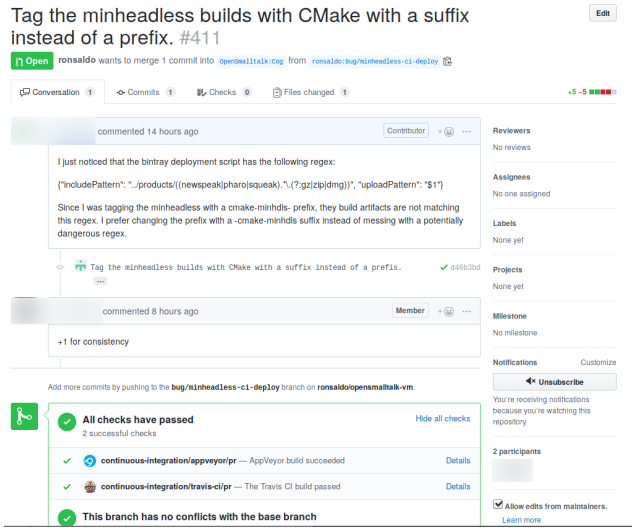


Figure 2.5: Discussion thread for a pull request. This is pull request number 411 or PR number 411 of the OpenSmalltalk/opensmalltalk-vm project on GitHub. This pull request has two discussion comments. In the lower section, there are links and checkboxes with the results from two different continuous integration servers.

The contributor pushes his or her changes of a public version of the forked repository belonging to his or her GitHub account.

2. The contributor uses the GitHub web interface for creating a Pull Request to the upstream repository[2] of the open source project. See Figure 2.4 for an example of the pull request creation form.
3. The changes of the pull request can be tested by a continuous integration server[3] if the open source project has support for continuous integration. For an example of this, see Figure 2.5.
4. The integrators of the open source repository receive the pull request through GitHub.

the fork. Forks can be integrated into the mainstream project, or they can become a completely new project and open source community.

[2]Upstream repository means the main public and visible source code repository of a software project.

[3]A continuous integration or CI server is a machine that automatically downloads commits from a sofware repository, compiles the software and runs tests.

5. The pull request is reviewed by the open source project integrators.
6. The pull request can be publicly discussed by the open source project integrators, the pull request contributor, and the open source project community in general. See Figure 2.5 for an example of these discussions.
7. The pull request contributor can make additional commits to the pull request in order to comply with additional request and requirements imposed by the software project integrators.
8. Once a pull request is reviewed and discussed, it can be accepted or rejected by the software project integrators by just clicking a button on the GitHub web interface. See Figure 2.6 for an example of how these buttons look.

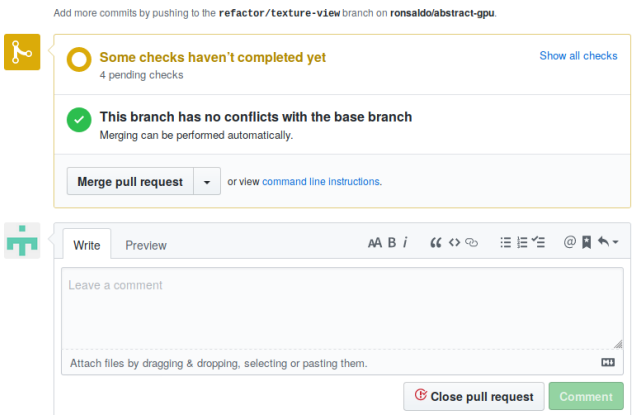


Figure 2.6: Buttons at the end of the GitHub pull request interface.

2.3 Source code metamodeling

The construction of a visualization for software source code requires the previous construction of a model for the software source code. The constructed software source code model is the instance of a metamodel. A metamodel is a model of a model [20]. The usage of a metamodel for analyzing source code allows one to abstract far-away from the textual oriented source code strings that are difficult to analyze, by representing the software in terms of objects.

For example, a simplistic metamodel for software can be constructed by having elements with the following names: packages, classes, meta-classes, methods, functions, global variables and objects. These elements are connected by the following relationships:

- Packages can contain classes, global functions and global variables.
- Classes can contain methods.
- Objects are instances of classes.
- Classes are also objects, so they are an instance of a meta-class.
- A global variable is a named location for holding an object.

A metamodel with this description can be used for representing a wide variety of source code in different languages such as Smalltalk and Python. Source code written in these

languages can be parsed and used for constructing instances of this simplistic meta-model. For the implementation of Git Thermite, we constructed a new metamodel similar to the one of this example.

Ring is the standard metamodel for Smalltalk source code [21] that is used by the Pharo runtime. The Pharo runtime uses internally Ring for implementing its source code browsers, and for implementing the Monticello version control system (See Section 2.1.3). For this reason, in our tool we reuse Ring for constructing an initial metamodel instance when analyzing Pharo source code. We then convert this Ring metamodel instance into an instance of the metamodel that we designed and constructed for the internal usage of our tool.

2.4 Software structural visualizations

This section of the state of the art describes the following visualizations for the structure of a software: Polymetrics Views in Section 2.4.1, CodeCity in Section 2.4.2, and CuboidMatrix in Section 2.4.3. CodeCity and CuboidMatrix are two visualizations that were evaluated with a controlled experiment. We are using the controlled experiments of these two works for basing the design of our controlled experiment in Chapter 6.

2.4.1 Polymetrics views

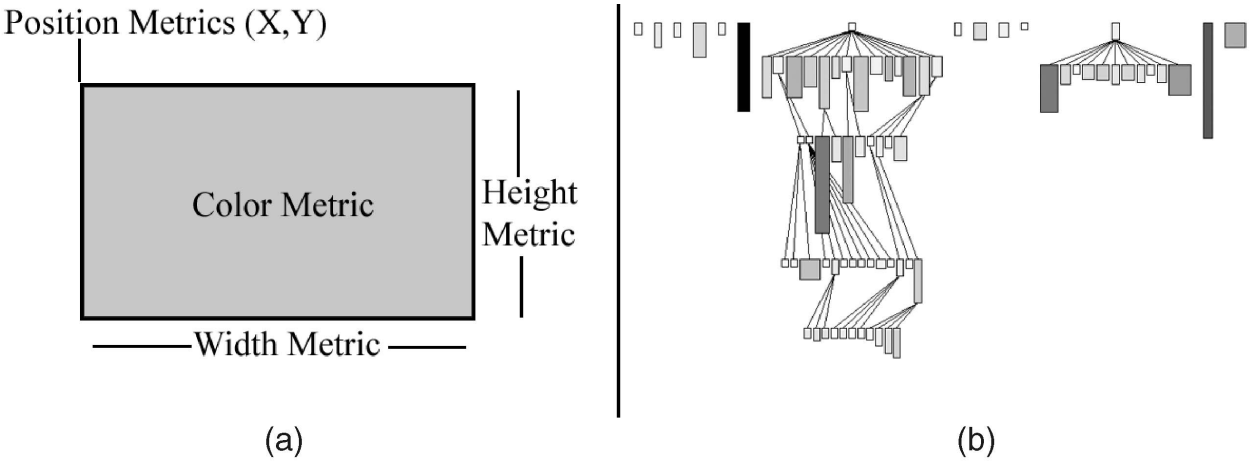


Figure 2.7: Example of a polymetric view (Image from [1]). Part (a) shows how a single node can represent up to five different metrics [1]. Part (b) shows a *system complexity view* with nodes representing a class hierarchy on a tree layout [1]. The metrics used in this visualization are: the number of class attributes on the width, the number of methods on the height, and the color shade represents the number of lines of code per class [1].

Polymetrics views by Lanza *et al.* [1] is a 2D visualization for reverse engineering software. As described by Lanza *et al.*, these polymetric views are:

"A lightweight approach based on *the combination of software visualization and software metrics*, by enriching visualizations with metrics information" [1]

In a polymetric view, a single rectangular element can encode up to five metrics (See Figure 2.7 part (a)). These metrics can be encoded by using the position of the element (x and y coordinates), the dimensions of the element (width and height), and the color of the element. Figure 2.7 part (b) includes an example for a polymetric view that combines a structural software visualization for a class hierarchy with different metrics for each class. The class metrics are represented in this visualization by using the dimensions (width and height) and color.

2.4.2 CodeCity

CodeCity visualization. CodeCity [2, 22] is a 3D software visualization based around the city metaphor (See Figure 2.8). This visualization represent classes of a system by using *buildings*. Packages are represented by grouping *buildings* in *neighbors*. Like the case with polymetrics views (See Section 2.4.1), different metrics of a class can also be represented on this visualization by using: the *width*, the *height*, the *depth*, and the *color* of a building.

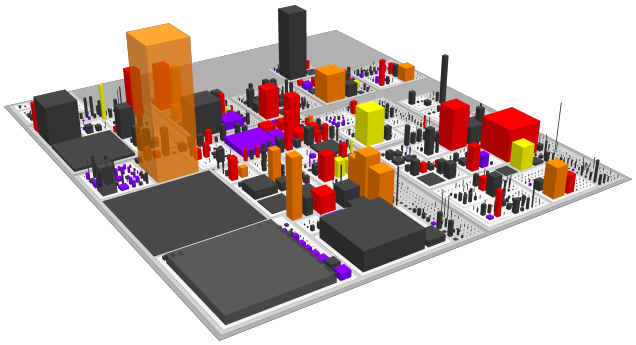


Figure 2.8: CodeCity software visualization (Image from [2]). This visualization maps the number of methods of classes to the *height* of the building. The *base size* (width and depth) is mapped to the number of lines of code of each class. The color of these classes is used to display *design problems* such as: brain class (yellow), god class (red), a class that is both a brain and god class simultaneously (orange), data class (purple). This particular CodeCity software visualization is called a *disharmony map* [2] because of the presence of this last metric (design problems).

CodeCity controlled experiment. The CodeCity visualization was evaluated with a controlled experiment [2, 22]. This controlled experiment consisted on comparing CodeCity against a *baseline tool* that is composed by the simultaneous usage of two tools: the Eclipse IDE for extracting data from Java projects, and Excel for analyzing this data. The pilot experiment[4] of CodeCity involved the participation of Master students of the University of

[4]A pilot of a controlled experiment is a run with few participants that is used for testing the experiment itself. The participants of a pilot experiment cannot be reused in a future experiment because they have learned to use the tool, and that can introduce a bias.

Lugano [2]. The full experimental run for testing CodeCity involved participants coming the *academia* and the *industry*, with participants coming from three different cities in Switzerland, Belgium and Italy. In total, this experiment had 44 participants [2].

Controlled experiment result. The results of the CodeCity controlled experiment are differents for very focused tasks that involved finding a precise answer, in comparison with tasks that involved getting an overview of the code. In the case of very focused tasks, Eclipse and Excel was better than CodeCity, however CodeCity for most of these tasks "managed to be on par to Excel" [2]. In the other case, "tasks that benefit from an overview, CodeCity constantly outperformed the baseline" [2].

Limitations. The main limitation we have with CodeCity is that we cannot apply this visualization to represent software change sets directly. For this reason, we are not reusing this visualization in our tool. However, we are using the controlled experiment that was used to validate CodeCity [2, 22] as a base for designing our own controlled experiment for validating our tool (See Chapter 6).

2.4.3 CuboidMatrix

CuboidMatrix visualization. CuboidMatrix by Schneider *et al.* [3] is a software visualization based around 3D space time cubes (See Figure 2.9 for an example). This software visualization is actually an application of another 3D visualization called MatrixCubes by Bach *et al.* [23] that was designed to visualize the evolution of networking connection graphs [23] along the time. This visualization uses two axes (X and Y) to represent the existence of a relationship between two elements during a specific time slice which is represented by the other axis (Z). In this way, this visualization can be used to represent the changes on the existence of relationships between elements. The original versions represents the existence of connections between pairs of elements in a computer network [23]. The CuboidMatrix software visualization is used to represent two different kinds of software comprehension data: the interaction between classes along the execution of a program (See Figure 2.9 for an example of this); and the evolution in the application of Lint[5]-like rules over several versions in the history of a software [3].

CuboidMatrix controlled experiment. CuboidMatrix was evaluated with a controlled experiment [3]. This controlled experiment compared CuboidMatrix against Excel as a baseline tool [3]. This controlled experiment involved two software comprehension tasks on class interactions during a program execution. These two tasks used two different datasets D1 and D2. During this controlled experiment, each participant answered a set of question about dataset D1 by using one tool (CuboidMatrix or Excel), and then they answered the same

[5]Lint is a software source code static analysis tool for detecting programming error, and software quality issues. This tools works by applying a series of pattern matching rules on the source code.

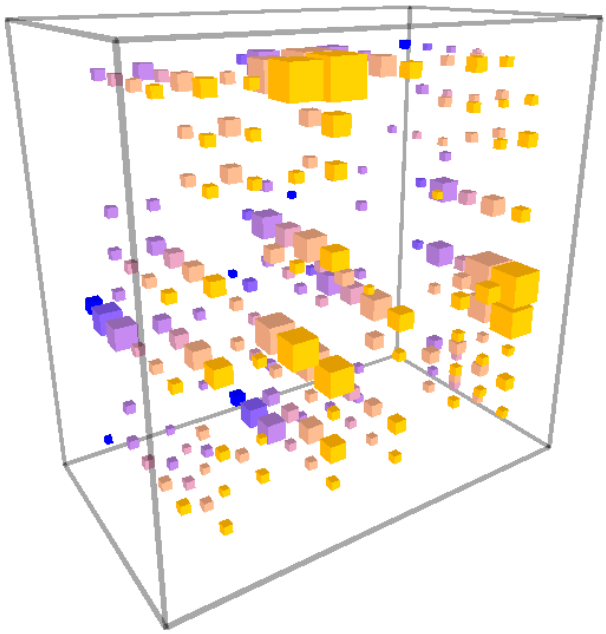


Figure 2.9: CuboidMatrix visualization (Image from [3]). This visualization represents the interaction between classes along the execution of a program. The Z axis represents time, and the X and Y axis are used for labeling classes. The presence of a cube indicates that a pair of classes (X and Y axes) interacted during a given time frame (Z axis). The size of a cube represents the number of interactions, and a color gradient from blue to yellow is used to represent the duration of these interactions.

question about dataset D2 by using the other tool. The pilot study for this controlled experiment had two participants [3]. The execution of this controlled experiment had a total of eight participants:

"Among these participants, there were 3 professional software engineers, 2 PhD students, 2 master students, and 1 undergrad student." [3]

CuboidMatrix controlled experiment results. The "participants performed significantly better using CuboidMatrix than using Excel" [3]. The conclusion by Schneider *et al.* is that for their tested "dataset and questions, CuboidMatrix significantly outperforms Excel." [3]

CuboidMatrix case study. The usage of CuboidMatrix with software quality rules was used to perform a case study on the history of a large software [3]. This case study found limitations in terms of visualization scalability because they were analyzing a large dataset that produced a visualization composed of tiny elements so small that non-differences between them are seen [3].

2.5 Software change set visualizations

2.5.1 Torch

Torch [4, 5] is a dashboard for analyzing visual source code changes in Pharo (See Figure 2.10). Torch can visualize arbitrary commits and change sets made in Pharo with the Monticello version control system. The Torch dashboard is composed of the following panels:

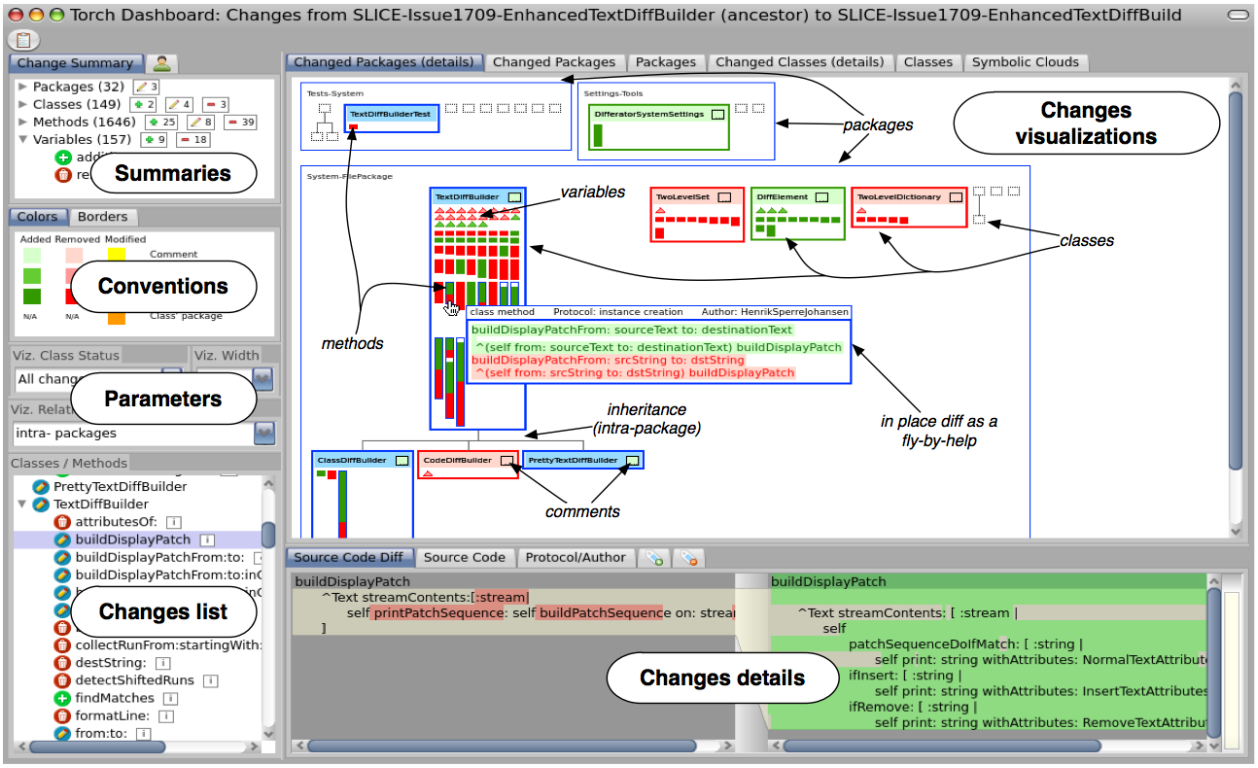


Figure 2.10: The Torch dashboard (Image from [4, 5]).

- Summaries: on Figure 2.10 top-left there is the *summaries* panel. This panel displays a list with the number of entities (packages, classes, methods and variables) that are impacted by the analyzed change sets. A summary with the number of added, removed and modified is available per entity [4, 5].
- Conventions: this panel on the middle top-left of Figure 2.10 displays a legend with the *color* conventions that are used in all of the visualizations [4, 5].
- Parameters: this panel on the left side of Figure 2.10 contains a set of controls for selecting different filters and parameters for controlling the visualizations of the change set [4, 5].
- Changes list: this panel on the bottom left side of Figure 2.10 displays a tree view with a list of all the classes and methods in the change set. An icon on the left side of each element in this tree view indicates whether the element is added (green + icon), removed (red trash can icon) or modified (blue pencil icon).
- Changes details: this panel on the bottom of Figure 2.10 contains a textual diff for a selected method or class definition.

- Changes visualizations: this is the main panel of the Torch dashboard. This panel can be seen on the top right of Figure 2.10. Several visualizations of the change set can be displayed on this panel by selecting them by clicking on one of the tabs above this panel. These visualizations are variations on three kinds of visualization: the *structural changes visualization* describing the changes to classes, methods and instance variables; the *condensed visualization* of class changes; and the *symbolic cloud* visualization describing changes on the vocabulary used by the software developers.

Structural and condensed changes visualization. The two visualizations of Torch for displaying changes to classes are very similar. The *structural changes* visualization displays the changes to class methods and instance variables as separate elements inside of a rectangle with the class new on the top. For an example of the structural changes visualization, see the three classes (DiffElement, ScreenController and PositionableStream) on the top side of Figure 2.11. In the case of the *condensed visualization* classes, methods and instance variables represented as a single merged element. For an example of the condensed visualization, see the same classes of the previous case, but on the bottom side of Figure 2.11.

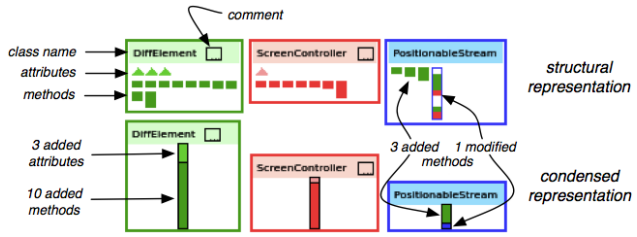


Figure 2.11: The Torch structural and condensed changes visualizations (Image from [5]).

Visualization elements. The structural changes visualization of Torch is composed of the following elements:

- Packages. Packages are represented as large rectangles with a blue border. The name of a package is displayed on the top left corner of each one of these rectangles. For example, Figure 2.10 contains the following packages: *Tests-System*, *Settings-Tool* and *System-FilePackage*.
- Unchanged classes. Unchanged classes are represented as small squares with a dotted border. For example, on the left side of the *Tests-System* package in Figure 2.10 there is an unchanged class hierarchy with a parent superclass and two children subclasses.
- Classes with changes. Classes with changes are represented as large rectangles inside of packages with a colored border. The color of the border indicates whether the class is added (green), removed (red), or modified (blue) [5]).
- Instance variables or attributes. Class attributes are represented with colored triangles inside classes. The color of these triangles indicates if the attributes is added (green) or removed (red) [5].
- Methods. Methods are represented as colored rectangles inside of classes. These colored rectangles are a visual representation of the method textual diff. Each one of these rectangles has the same width, and the height corresponds to the number of lines of

code in the method. The Y coordinate on these rectangles is mapped to each line number of the method. Lines that are *added* are represented with a *green* color, and lines that are *removed* are represented with a *red* color. *Unchanged lines* are represented with *white*. *Modified methods* in addition have a *blue border* [5].

Symbolic cloud visualization. The symbolic cloud visualization is an application of the *Word cloud* [24] visualization to the symbols that are used on the program source code. This visualization represents the usage of *words* along with their *usage frequency*. This visualization arranges these words by assigning a different size to each word. Words with a larger usage frequency (*i.e.,* words that are most often used) are assigned a larger size than words with a smaller usage frequency (*i.e.,* words that are less often used). The intention of this symbolic cloud is on displaying the changes on the vocabulary that is used by the developers of the software [4, 5].

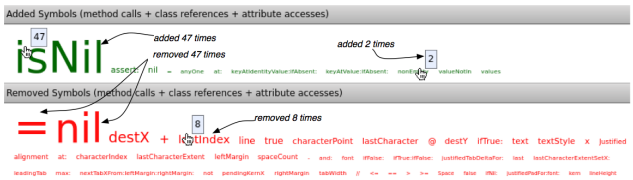


Figure 2.12: The Torch symbolic cloud visualization (Image from [5]).

Omnipresent contextual diff. An important concept that is present on the Torch visualizations is the concept of an *omnipresent contextual diff*. This concept implies that is possible to obtain the actual textual diff for any element that is represented on a visualization. In Torch, this *omnipresent contextual diff* is achieved through the implementation of a *colored textual diff* on a *fly-by-help*. This *fly-by-help* is a panel with a colored textual diff that appears automatically when the cursor is hovering an element in the visualization. An example of this *fly-by-help* can be seen in the middle of the visualization panel that is present in Figure 2.10.

Torch Evaluation. The evaluation of Torch was conducted through two case studies. Torch was not evaluated with a controlled experiment because finding a large number of integrators working on large systems is difficult [4, 5, 25]. These two case studies were:

1. A field study with six integrators of three different open source Smalltalk projects [4, 5, 25].
2. A pre-experimental user study with ten participants which are mostly Smalltalk developers [4, 25].

Limitations. Since Torch was implemented for analyzing commits in Monticello written Pharo, Torch cannot be used for analyzing commits in Git, or commits made in other languages. This is the main limitation in the implementation of Torch. Another limitation

of Torch is the lack of maintenance which implies that Torch cannot be used on the latest versions of Pharo.

Due to the similarities of Torch with the work of this thesis, the main visualization designed for this thesis is heavily based on Torch. Since the limitations of Torch are related to its implementation, and they are not limitations because of the design of Torch itself, we are not attempting to make a tool better than Torch. In fact, if we were attempting to make a tool better than Torch, then we should be comparing our tool against Torch in our controlled experiments.

2.5.2 Motive

Motive [6] is a tool for visualizing software change sets for Java. The visualizations generated by Motive are UML diagrams. For an example of how the Motive visualization tools looks, see Figure 2.13. The Motive visualization tool is composed of three panels:

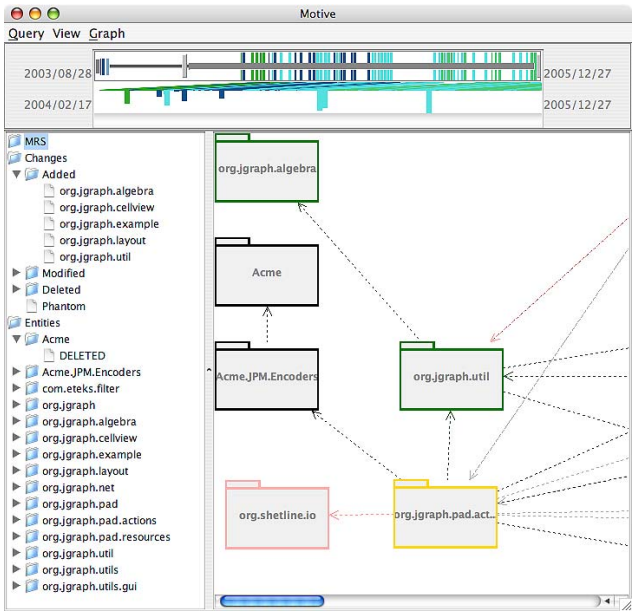


Figure 2.13: Motive software visualization (Image from [6]).

1. Temporal Slider (See Figure 2.13 on the top): these are two sliders visualizing an overviews of the changes present in the the change set. The top slider represent all of the changes in the analyzed source code repository. The bottom slider represents all of the changes during a selected time period for analysis.
2. Hierarchical Summary (See Figure 2.13 on the left): this is a tree view that displays a detailed list of the changes present in the change set.
3. Graph View (See Figure 2.13 on the right): this is a visualization of the final effects of applying a change set by using UML diagrams of classes and packages.

The evaluation of the Motive tool was performed through the conduction of two *case studies* [6].

2.6 GitHub visualizations

GitHub visualizations. This section describes other visualizations that are found on GitHub. GitHub is an important social work environment for open source projects, communities and developers. We are including these visualizations because they are integrated in the GitHub web platform.

GitHub contribution matrix. The GitHub contribution matrix (See Figure 2.14) is a visualization that displays the number of contributions per day of a software developer. This visualization is found on the GitHub profile web page for a GitHub user. This visualization is an application of a data visualization known as *Heatmap* [26].

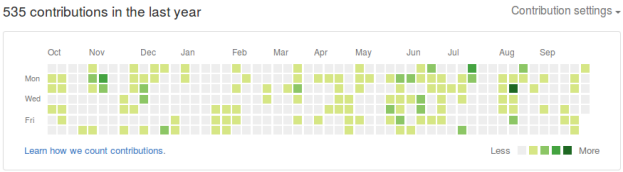


Figure 2.14: GitHub contribution matrix.

GitHub fork graph. The GitHub fork graph (See Figure 2.15) is visualization describing the *forks* of an open source project repository. This visualization answers the following questions about a fork: how many forks on GitHub exist? who has made a fork? at what commits the forks are branching off?

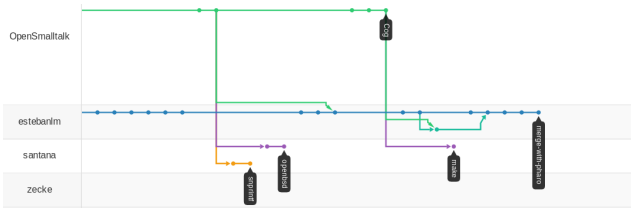


Figure 2.15: GitHub fork graph

GitHub repository contributions. The GitHub contributions charts (See Figure 2.16) are a series of line plots describing the commits that are made by indivual contributors to a particular open source project on GitHub. The line plots map the number of commits to the Y axis, and the time the commits are made to the project on the X axis.

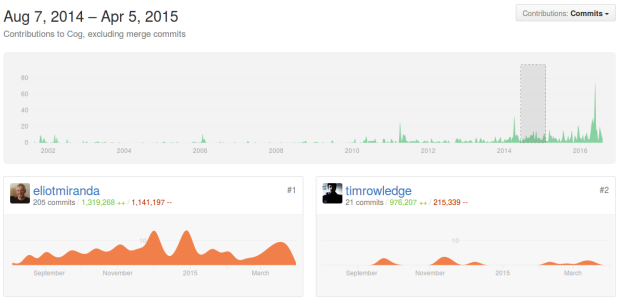


Figure 2.16: GitHub chart of contributions to a repository.

Chapter 3

Methodology

This chapter describes the methodology followed by this thesis. This methodology is an iterative process whose steps are summarized by the flow diagram in Figure 3.1. Each section present in this chapter corresponds to a step in this flow diagram.

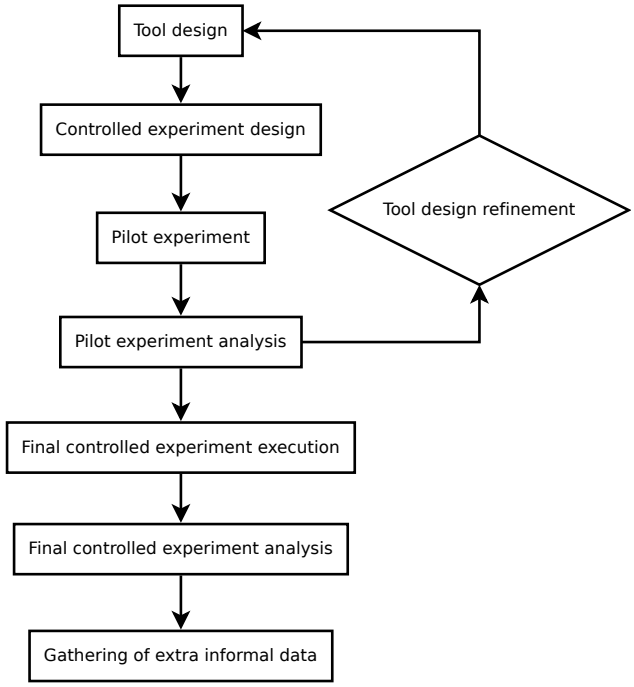


Figure 3.1: Methodology flow diagram

This methodology is the application of a trial and error method for designing the final tool that is ultimately tested by this thesis. This trial and error method resulted in four different visual tool design iterations. These preliminary design and testing iterations are described in coarse detail in Chapter 5. A detailed description of the final designed tool is present in Chapter 4.

3.1 Tool design

The visual tool for analysing change sets is designed during this step in the methodology. This step may be revisited after receiving negative feedback, or the detection of serious design flaws during the execution of a pilot experiment.

3.2 Controlled experiment design

A controlled experiment is an empirical method used in software engineering research [27]. The objective of a controlled experiment is to measure the relationship between dependent and independent variables. In our case, the independent variable is the tool used for performing a source code change set analysis. The dependent variables that we are particularly interested are some metrics of the change set analysis process: time of analysis, precision of the analysis, difficulty of doing the analysis, etc.

After designing a version of the visual tool that is apparently good enough for testing, the next step in the methodology is the design of a controlled experiment, or the refinement of a previous iteration of the controlled experiment, if it exists. The design of the controlled experiment involves the following three main elements:

- The design of a form with the questions of the experiment itself.
- The preparation of learning material to teach the test subjects on how to use the tool.
- The selection of commits, pull requests, or source code change sets to test in the experiment. This is probably a strong source of experimenter induced bias. In our case we chose to make this bias explicit by describing the exact public commits used in our controlled experiments.

The most important part of this section is the design of the form with the questions that are going to apply to the test subjects. This form is composed of the following sections:

1. Questions about the practitioners.
2. A learning section where the practitioner has to read the learning material and he has to experiment and practice with the tool.
3. An experiment with unfamiliar code. In this section the practitioner has to answer some questions about change sets that the experimenter selected for each participant, by using the visual tool and the baseline tool. The order of the tools and the change sets used in this section are randomized for the different participants, in order to reduce learning bias. The baseline tool chosen for this experiment is the *textual diff* tool because of its ubiquity for comparing source code change sets.
4. An experiment with familiar code. This section is the same as the previous section, but it is the participant who chooses the software to analyze. Ideally, the participant knows or has some expertise in the software to analyze in this section.
5. General impressions and feedback. This last section asks the participant questions about his general impression of the tool during the experiment, and it also gives the

participant an opportunity to give feedback.

6. Application of the NASA-TLX [28] form, a widely used test for measuring cognitive load during a workload. We asked participants to fill two copies of this form at the end of the experiment: one for the visual tool, and one for the textual diff tool.

The main target demographics for the visual tool is composed by practitioners in the software development industry, with an emphasis on developers that participate in open source projects. The main purpose of the visual tool is on facilitating the labor of a software integrator. They are the people that review contribution to a software project before deciding the acceptance or rejection of a change set. Another purpose of this visual tool is on helping individual contributors in reviewing their own changes before writing a commit message, or before submitting contribution for a reviewing and approval process. Since the main target demographics is composed by people fulfilling these two roles, it is unacceptable to use undergrad students without software development experience as participants of a proper controlled experiment. These people without experience can only participate in pilot experiments. For the proper controlled experiment, we selected people that belongs to at least one of the following groups:

- Professional practitioners in the industry.
- Professional practitioners in an open source community.
- PhD students and people with a completed Phd.

Unfortunately, finding many people with an integrator role in an open source project willing to participate in a 3 hours controlled experiment is a difficult task. For this reason, we selected people from these groups for the execution of the main controlled experiment.

More details of the controlled experiment design are presented in Chapter 6.

3.3 Pilot experiment

The immediate step after the design of a controlled experiment is to test the tool and the experiment design itself by conducting a pilot. A pilot experiment is conducted with few participants: no more than two or three participants. The participants of a pilot experiment cannot be reused in a posterior full experiment execution. The main objective of the pilot experiment is to detect small design errors in the tool, or in the experiment itself. The participants of a pilot experiment cannot participate again in a future full controlled experiment. Undergrad students are allowed to be participants in a pilot experiment, but they are not allowed to be participants of the full controlled experiment.

3.4 Pilot experiment analysis

After conducting a pilot experiment, the following step consists in analyzing its results. The results of a pilot experiment, or feedback gathered from the participants can show some

serious design flaws in either, the visual tool design, or the controlled experiment design.

If there are important problems in the controlled experiment design, then this feedback is used to improve on the experiment design, and another short pilot is conducted, until the experiment design is deemed good enough for conducting a final experiment with participants.

If problems are detected in the visual tool design, then the tool design is improved by going back to the first step of the methodology. After improving the tool, there is usually the need to also improve on the controlled experiment design, so another pilot experiment is required.

If there are only small design issues on either the visual tool, or the controlled experiment design, these small issues are then fixed. After fixing these small issues, we proceed to the final experiment execution.

3.5 Tool design refinement

When important design flaws are detected on the visual tool during the execution or analysis of a pilot experiment, it becomes necessary to fix these design issues. The existing tool design is improved by implementing features that are suggested by participants, or by implementing features that solve or help solve a critical problem detected during experimentation.

If the features added or fixed during this stage are major, they involve a restart of the methodological work flow in terms of having the necessity of improving the controlled experiment design, and redoing the pilot experiments. On the contrary, if these changes are minor there is no need to redo the complete experiment design.

3.6 Final controlled experiment execution

After having a visual tool design that is good enough (*i.e.,* does not have obvious design flaws detected on a pilot experiment), and a controlled experiment design that is good enough, then comes the execution of the final controlled experiment with people.

The final controlled experiment of this thesis was conducted with five Pharo practitioners. Each participant during the final experiment analyzed:

- Two commits of familiar code using the visual tool.
- Two commits of familiar code using the baseline tool.
- Two commits of unfamiliar code using the visual tool.
- Two commits of unfamiliar code using the baseline tool.

In the case of trivial, or change sets with a very small number of lines of code (*e.g.,* one to ten lines) we believe that our visual tool does not provide an advantage above a textual

diff.

The selection of the commits for unfamiliar code is made globally with the objective of allowing comparing the results of participants per commit. The tool and order of analysis of these commits is randomized for each participant. The randomization of the commit analysis order is made with the objective of reducing participant learning bias. The randomization of the tool used for analyzing each one of these commits is made with the objective of having a similar number of answer with each different tool for each commit. No commit is analyzed multiple times by any participant. The commits used for the final controlled experiments are the same commits that are selected for the pilot experiment performed previous to the final controlled experiment. This pilot is descibed in Section 7.2, and the commits are specified in Table 7.1.

The selection of familiar code for each participant is made by the experimenter at experimentation time. The participants has to suggest a project to analyze before or at the beginning of the experimentation sessions. The familiar code commits are selected with the criteria of having many lines of code in the order of between 500 and 5000 lines of code in the case of a textual diff, or having several changed elements in the case of a visualization.

The final experiment for each participant had a duration that was between one and three hours. More details about the final experiment execution are presented in Section 7.3. These extra details include information about who are the participants, how they were selected, and how the change set that they have to analyze are selected.

3.7 Final controlled experiment analysis

The analysis of the data gathered by the final controlled experiment is the most important part of the methodology of this thesis. This part involves testing the actual hypothesis or even formulating some new ones.

The results obtained during the final experiment with Pharo practitioners are presented in Section 7.4. We discuss these results and further analyze them in Chapter 9. In particular, there are no qualitative differences in terms of time, or analysis performance between using the visual tool or using the baseline tool. However, there is an apparent positive user experience perception in favor of using the *visual tool*. This apparent positive perception is supported by the written feedback given by the participants at the end of experiment, and by their responses on filling the NASA TLX test form.

3.8 Gathering of extra informal data

An additional extra step to our methodology consists in gathering any extra informal data and feedback from the Pharo open source community. This extra informal data is obtained by contacting some of the contributors of this open source community directly and asking

them to test the visual tool and offer their impressions through a questionnaire.

This informal data is easier to gather than the data obtained by doing a controlled experiment, and it is also valuable by itself. This data can be used to support any additional hypothesis that comes from the controlled experiment result data without having to perform another controlled experiment.

We decided to add this additional step in our methodology because of the negative results that we obtained with our controlled experiment in terms of finding differences in: the time spent by an user analyzing a change set, the precision of the analysis, and in terms of inferring the the intention of changes. However, since we found an apparent positive user perception we wanted to gather extra data that support this hypothesis because our experiment was not designed to measure this variable in detail, and we are only measuring it through the NASA TLX responses, and the written feedback by the participants.

Chapter 4

Git Thermite

Git Thermite is the tool that we propose to solve the problem of facilitating the reviewing process of commits. This tools can be used to analyze a set of changes that belongs to one of the following types:

- The changes introduced by a single commit.
- The changes introduced between an arbitrary pair of commits.
- The changes introduced by a pull request.

The first section of this chapter (Section 4.1) provides basic instructions for installing Git Thermite, and instructions for constructing a visualization with Git Thermite. The remaining sections of this chapter provide a description for each one of the elements present in the Git Thermite tool.

4.1 Getting started with the tool

4.1.1 Installing Git Thermite

The Git Thermite tool is written in Pharo, a Smalltalk dialect. We are making Git Thermite available on the following public GitHub repository: `https://github.com/ronsaldo/pharo-git-thermite`

Installing Git Thermite requires a running Pharo 7 image which can be obtained at the official web site of the Pharo project: `https://pharo.org/` Once a Pharo 7 image[1] is downloaded and opened, the main Pharo desktop is opened. By clicking anywhere on this desktop with the left button of the mouse the Pharo *World menu* is opened (See Figure 4.1 Part A). Installing Git Thermite requires opening a *Playground* (See Figure 4.1 Part B) through the *World menu*. Once a *Playground* is opened, the Git Thermite installation script has to be

[1]Pharo is an image based system. In this kind of system, there is no notion of traditional source code files. An image holds a memory dump of the whole system with all of its objects.

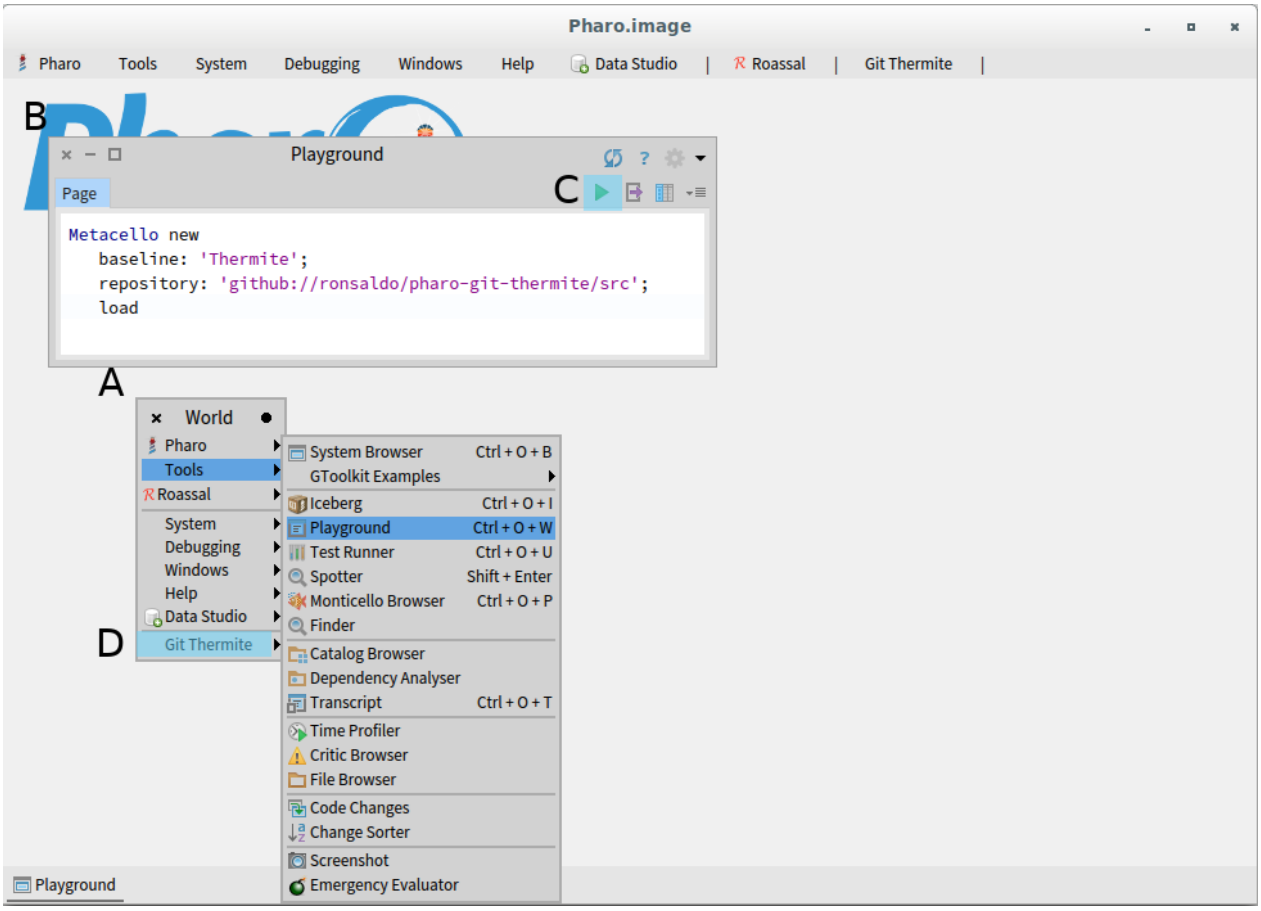


Figure 4.1: Pharo image desktop. Part A is the *World menu*. Part B is the *Playground* opened with the world menu. Part C is the *Do it* button for executing the Pharo script in the *Playground*. Part D is the *Git Thermite* menu that is added after installing Git Thermite on the Pharo image.

copied to the *Playground* and executed by clicking on the playground *Do it* button (See Figure 4.1 Part C). Once Git Thermite is installed on the Pharo image, additional Git Thermite related entries are added to the *World Menu* (See Figure 4.1 Part D, and Figure 4.2). Listing 4.1 contains the Git Thermite installation script, which is also available on the GitHub web page.

Listing 4.1: Git Thermite Pharo installation script.

```
Metacello new
   baseline: 'Thermite';
   repository: 'github://ronsaldo/pharo-git-thermite/src';
   load
```

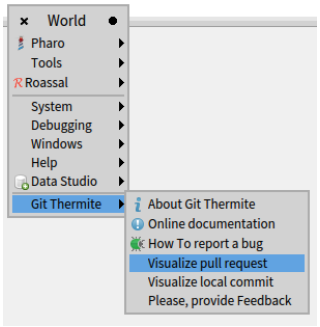


Figure 4.2: The *World menu* entries that are added by the Git Thermite installation.

4.1.2 Visualizing pull requests

For visualizing a GitHub pull request, there is a dialog for constructing a pull request visualization. The dialog in Figure 4.3 can be accessed through the Pharo *World menu*. This dialog requires entering: the name of the GitHub project owner; the name of the GitHub project; the number of the pull request; and selecting the programming language of the source code to analyze.

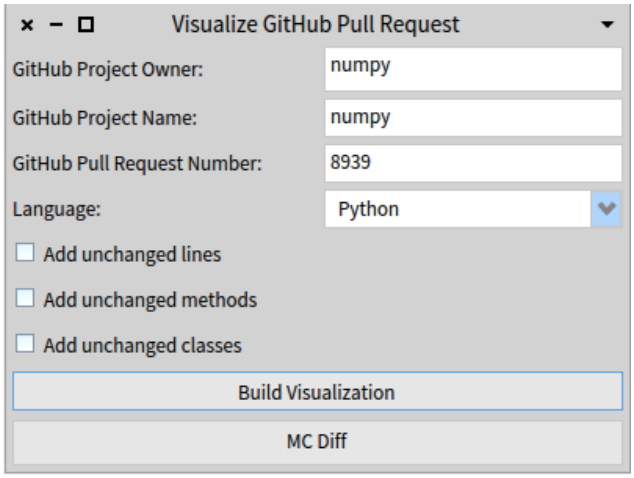


Figure 4.3: Dialog for starting a pull-request visualization.

4.1.3 Visualizing local git commits

For visualizing an arbitrary commit from a local git clone, it is possible to use the dialog present in Figure 4.4. Like the dialog for visualizing a pull request, this dialog can also be accessed through the Pharo *World menu*. This dialog required entering: the path of the local git clone; the name of the target commit; the name of parent commit used to compare the target commit; and selecting the programming language of the source code to analyze.

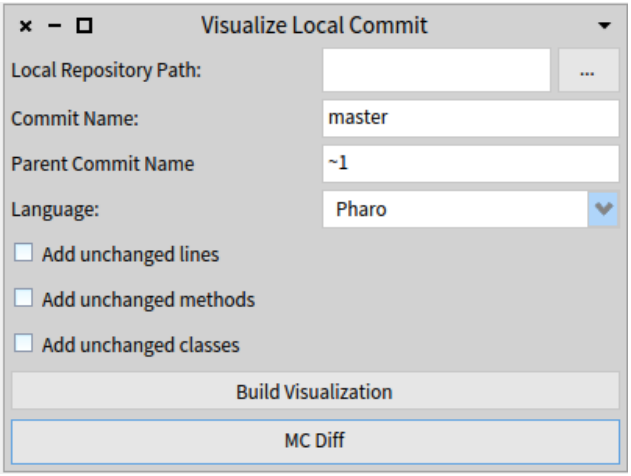


Figure 4.4: Dialog for starting a local commit visualization.

4.1.4 Monticello integration

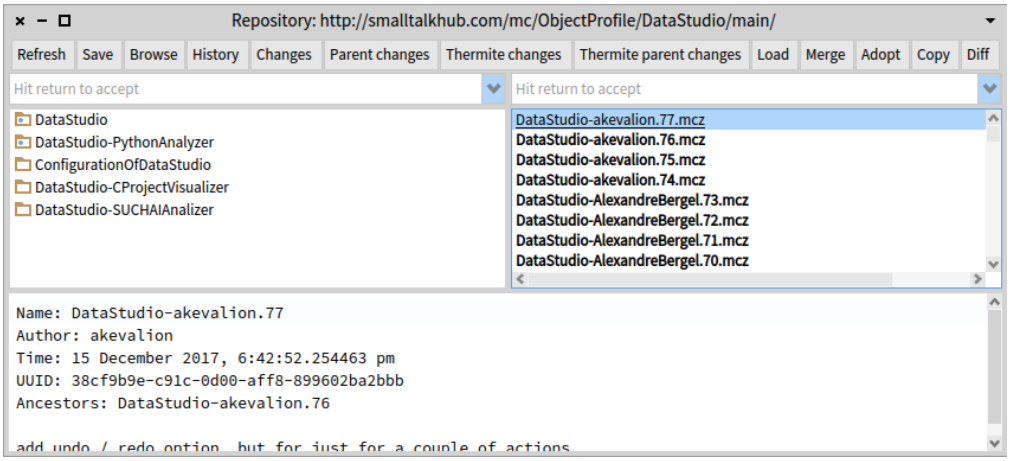


Figure 4.5: Git Thermite integration on the Monticello browser. The *Thermite changes* button allows comparing the selected commit with the working copy in Pharo image. The *Thermite parent changes* button allows comparing the selected commit with an arbitrary ancestor commit. This ancestor commit is selected from drop-down list that appears after clicking on the *Thermite parent changes* buttons.

Monticello is the traditional source control system that is included in Pharo. To facilitate the visualization of commits in Pharo made with Monticello, we integrated Git Thermite with Monticello by extending the Monticello browser with buttons to generate visualizations with Git Thermite (See Figure 4.5). The commits used in the final Pharo controlled experiment presented in Chapter 7 are extracted by using this Monticello integration mechanism. For a detailed explanation about how Monticello is used in Pharo, refer to the official tutorial and documentation [17, 18]. Currently, Monticello in Pharo is being deprecated in favor of Iceberg, a Git and GitHub frontend written in Pharo.

4.1.5 Iceberg integration

Iceberg is the new official control version system available in Pharo. Iceberg is implemented as a frontend with Git and GitHub. We integrated Git Thermite into Iceberg before publishing our tool to the Pharo open source community, and before conducting the informal evaluation that is described in Chapter 8. The integration of Git Thermite with Iceberg is made in two places: the repository browser (See Figure 4.6), and the commit dialog (See Figure 4.7).

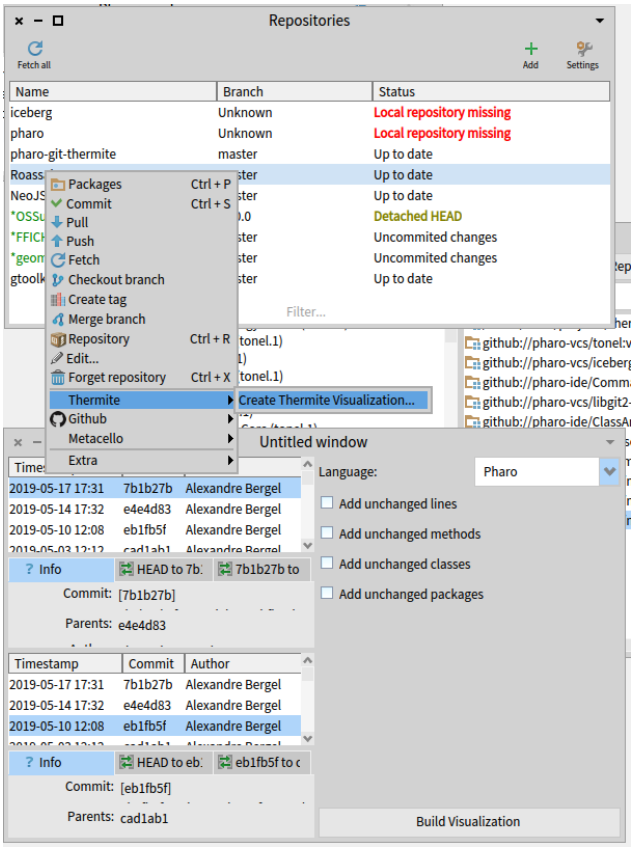


Figure 4.6: Git Thermite integration on the Iceberg repository browser.

The Iceberg *repository browser* is the main dialog that is presented by the Iceberg version control system. This browser presents a list of Git repositories that are registered in the Pharo image. The *add* button in this browser allows registering additional Git repositories. Right clicking on a repository on this list brings a contextual menu with several options (See Figure 4.6). On this contextual menu we added a *Create Thermite Visualization...* menu entry. This menu entry brings the dialog on the bottom of Figure 4.6 which allows selecting a pair of arbitrary Git commits, selecting filtering options, and selecting the programming language that is used on that Git repository. After these options are selected by an user, the user can click on the *Build Visualization* button for constructing a visualization with Git Thermite.

The commit dialog presented by Iceberg allows an user to review his or her changes, write the commit message, and then perform the actual commit. We integrated Thermite on this dialog by adding a *Thermite* button with magnifying glass icon on the top left corner of this

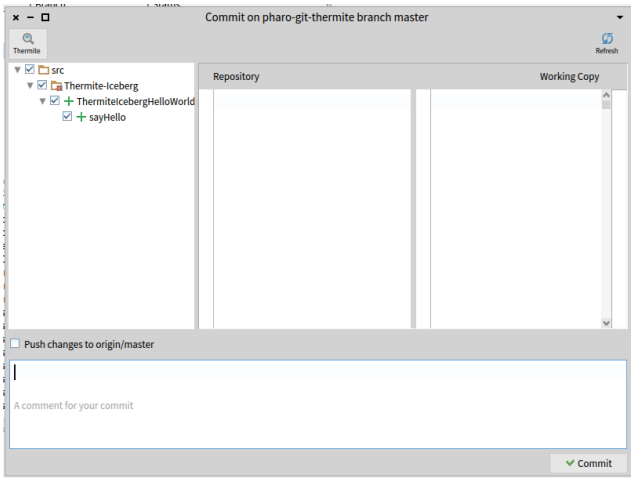


Figure 4.7: Git Thermite integration on the Iceberg repository browser.

dialog (See Figure 4.7). A Git Thermite visualization with the changes to be commited is constructed by just clicking on this button.

4.2 Color Legend

The color legend (See Figure 4.8) is composed of the following colors:

- Light-gray for unchanged elements.
- Green for added elements.
- Red for removed elements.
- Yellow for modified elements.
- Cyan for moved or renamed elements.

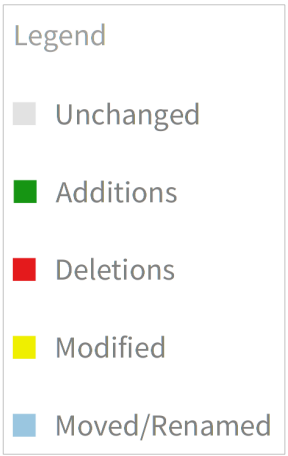


Figure 4.8: Color legend

These colors are used consistently by all of the visualizations present in our tool. The

selection of these colors responds to the following critieria:

- Colors are used for naming different categories. This means that any arbitrary choice of easy-to-differentiate colors can be used.
- Colors should be pleasant to see in order to reduce eye strain from long periods of tool usage. For this reason is preferable to avoid fully saturated colors.
- Standard textual diff tools are already using a green color for displaying added lines of code, and a red color for removed lines of code. For this reason, it makes sense to keep these two colors with the same meaning.
- Yellow and blue can both be used for displaying changed elements. However, some tools such as the Atom[2] text editor are already using the yellow color for representing modified files. This leave using blue for moved or renamed elements. In our case we used cyan which is ligher version of blue.
- The case of unchanged elements we are using a shade of gray because it represents a color without a hue. We used a light gray shade to contrast with the background.
- We also looked used the ColorBrewer[3] tool as a source of inspiration. This is a tool for choosing color palettes for visualizing data and making charts.

4.3 Unchanged source code elements and cluttering

An example of a visualization for a pull request without filtering unchanged, classes or methods is available on Figure 4.9. Most of the time, not doing this filtering introduces a lot of cluttering without adding relevant information in the visualization. For this reason, the default options of the tool is to exclude the unchanged elements from the visual report, and metrics charts completely. For this very reason, all of the examples presented during this chapter are filtering the unchanged elements.

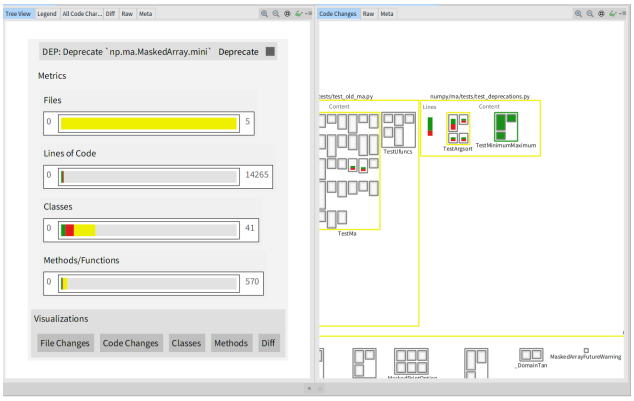


Figure 4.9: Visualization with unchanged lines, unchanged class and unchanged methods. This is *numpy/numpy* pull request number 8939.

[2]Atom: https://atom.io/
[3]ColorBrewer: http://colorbrewer2.org

4.4 Business Card Visualization

The first visualization (See Figure 4.10) presented by the tool is used to convey a high-level overview of the changes introduced in the commits that are being analyzed. This visualization is modeled after a *Business Card* metaphor with an identification of the changes, and presenting a high-level overview visualization in order to make a first impression of the changes. This metaphor is based around the purpose of a physical business card which is presented by a person to introduce himself or herself to another person. Like the case with the physical business card, our visualization business should answer the following basic questions: *what is the name of the change set?*, *how are can the changes be described in very coarse detail?* *how can I get more information about these changes?* In Figure 4.10 it is possible to see that this visualization is divided in three parts: the *title bar* (Part A), bar charts with *metrics* (Part B), and the *navigation buttons* (Part C).

Figure 4.10: Business card visualization for *numpy/numpy* pull request number 8939. This visualization corresponds to the same pull request that is visualized without filtering on Figure 4.9.

Title bar. The business card title bar (Figure 4.10, part A) is used to display metadata about the set of changes: the name of the set of changes; the type of the changes; and in the case of pull requests, whether automatic integration is possible or not.

The value of the title displayed in this view depends on the kind of change set: the commit message for a single commit; the name of the pull request when analyzing a pull request; the name of the two commits that are being compared in case the tool is being used to compare a pair or arbitrary commits.

The type of change is a single keyword that is used to give an initial impression on the

intention of the changes. This type is inferred heuristically by looking for some keywords in the commit messages, and the corresponding pull request discussion messages in the case of pull requests. The types that are inferred by using these heuristics are the followings:

- Bug fix (Bug keyword).
- New feature (Feature keyword).
- Enhancement (Enhance keyword).
- Deprecation (Deprecate keyword). This can be seen in Figure 4.10.

The possibility of automatic integration only applies for the analysis of pull requests. This data is extracted by using GitHub REST API, and is presented with a small square with one of the following colors:

- Green - It is possible to merge the pull request by just clicking a button in GitHub. GitHub determines this by creating a temporary merge commit.
- Orange - It is not possible to merge the pull request automatically. Conflicts have to be resolved manually by the integrator.
- Dark gray - The automatic merge possibility is not known, or is not reported by the GitHub API. This usually happens when analyzing pull requests that are already integrated. On newer versions of the Git Thermite tool, we are removing this square from visualization in this case with the purpose of reducing user confusions.

Hovering on a square with the cursor provides an additional tooltip with a description of the meaning of the square, and the meaning of its color.

Metrics. The metrics section of the business card (Figure 4.10, part B) is composed of colored bar charts. Each one of these colored bar charts displays how the changes are to the number of: files, lines of code, classes and methods or functions (*e.g.,* Python module functions).

Each one of these colored bar charts is composed by: the number zero on the left indicating the beginning of the scale; a set of the colored bars dis-aggregating the total number of elements; a number with the total of analyzed elements. The set of colored bars follows the standard color legend described Section 4.2: green for additions, red for deletions, yellow for modified, cyan for moved and or renamed elements, and light-gray for unchanged elements.

The colored bars in each one of these plots can be used to obtain additional information. By highlighting one of these bars with the cursor, a tool-tip is obtained with the number of elements represented by the colored bar. Clicking in one colored bar opens a visualization, that contains only the elements that are represented by the clicked bar. For example, clicking in the *yellow bar* of the colored bars representing *files* opens the file changes visualization (Section 4.5), with only the modified files, and excluding files that are completely added or completely removed in the change set. For the colored bars of the classes and methods/function charts, clicking opens the structural changes visualization (Section 4.6).

Navigation Buttons. In the last part of the business card (See Figure 4.10, part C) we have a bar with the navigation buttons. Each one of these buttons is used to open an additional view of the set of changes that is being analyzed. The actions of these buttons are the followings:

- File Changes: this button (Figure 4.11) opens the visualization with all of the file changes.
- Code Changes: this button (Figure 4.13) opens the visualization for structural changes with all of the packages/files.
- Classes: this button (Figure 4.17) opens the visualization for structural changes with all of the classes, by omitting the packages/files.
- Methods: this button (Figure 4.19) opens the visualization for structural changes with all of the classes all of the methods.
- Diff: this button (Figure 4.21) opens a view with textual diff for the whole set of changes.

4.5 File changes visualization

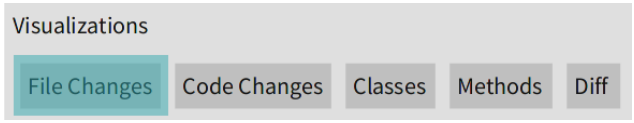


Figure 4.11: File changes visualization accessing button

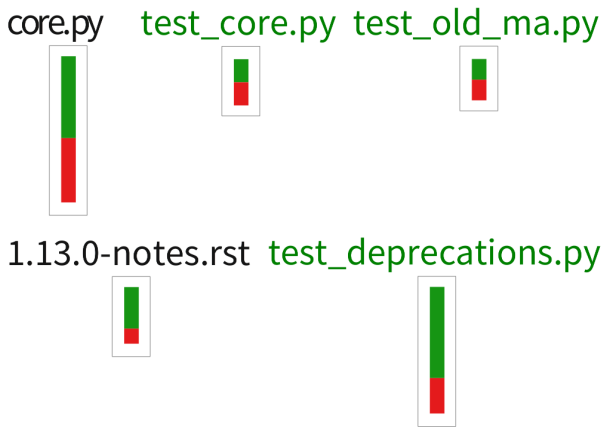


Figure 4.12: File changes visualization for *numpy/numpy* pull request number 8939. This corresponds to the same pull request that is visualized in Figure 4.10.

The file changes visualization describes the files that are added, removed or modified in the set of changes. In this visualization each file is represented with a box that contains the following elements:

- The name of the file above the box. If the name contains the word *test*, then the name label is colored with dark green.

- An embedded chart describing the number of lines that are unchanged (gray), added (green) or removed (red).

Moved files or moved lines or code are not represented on this visualization. The detection of moved lines of code is a difficult problem to solve because source repositories do not have a recording of movement actions by user. This means that source code movement has to be inferred from individual snapshots of the source code history through a heuristic approach. Since doing this properly is beyond the scope of this thesis, we are only doing it approximately by comparing the source code of methods by equality after trimming leading and trailing whitespaces. This code movement detection is only applied to the the visualization of structural changes.

Selecting these files with the mouse displays a tool-tip with the number of lines added, removed or that are unchanged. Clicking in one of these files opens a view with the textual diff of the file.

4.6 Structural changes visualizations

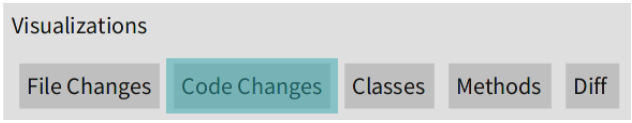


Figure 4.13: Code changes visualization accessing button

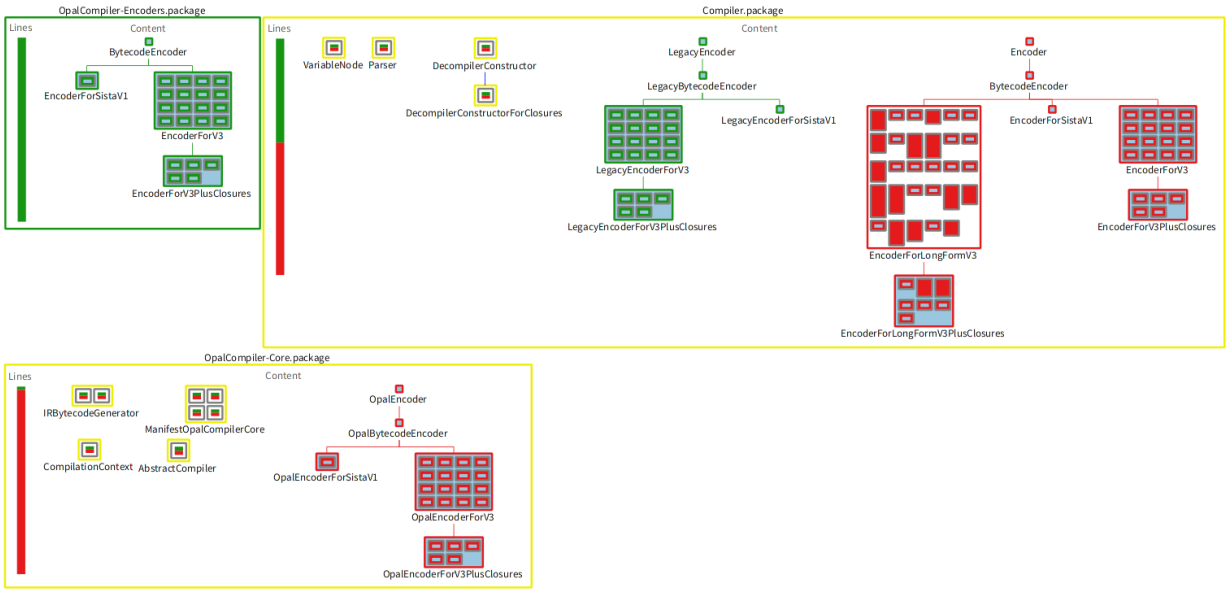


Figure 4.14: Code changes visualization for *pharo-project/pharo* pull request number 166.

The visualization for structural changes in Figure 4.14 is the main visualization provided by Git Thermite. The design of this visualization is based on Torch [4, 5, 21] by Gomez *et al.* From Torch we are taking its visualization based in boxes, and the very important concept of

having around a *contextual diff* (See Section 2.5.1). This concept means that it is possible to obtain the textual diff of the changes represented by each visual element in the visualization. This same structural changes visualization is reused when visualizing only classes, or when visualizing only methods.

Visual elements. The structural changes visualization is a hierarchical representation of the source code changes, organized around the following hierarchical levels: packages or files; classes; and methods or functions.

Packages or Files. Depending on the language, packages or files are the top-level containers, so they are represented as the big encompassing rectangles in the visualization. The border color of the package or file indicates whether the package is unchanged in light green, added in green, removed in red or modified in yellow. On the left side of the packages (and files), there is a bar chart with the total number of lines of code of the package that are added in green, removed in red, or are unchanged in light-gray.

Classes. Inside the packages or files, there are smaller rectangles representing the classes inside the package or file. Each one of these rectangles has the name of the class below the rectangle. The border color of the rectangle is used to indicate whether the class was completely added in green, completely removed in red, modified in yellow, or moved in cyan.

Lines joining classes are used to represent the *subclass of* relationship between classes. In other words, these lines are used to display the class hierarchies that are present inside of a package/file boundary, and how they change between commits.

Methods and/or Function. Methods and/or functions (*e.g.,* Python top-level functions) are represented with rectangles with a fully colored rectangle, or a rectangle with an embedded chart.

Interactions. Highlighting an element with the mouse in this visualizations displays a detailed tool-tip with a description of the element. For example (Figure 4.15), highlighting a method presents a tool-tip with the name of the methods, and number of lines that are added, removed or unchanged.

Textual diff. Clicking on an element opens a textual diff for the element (See Figure 4.16).

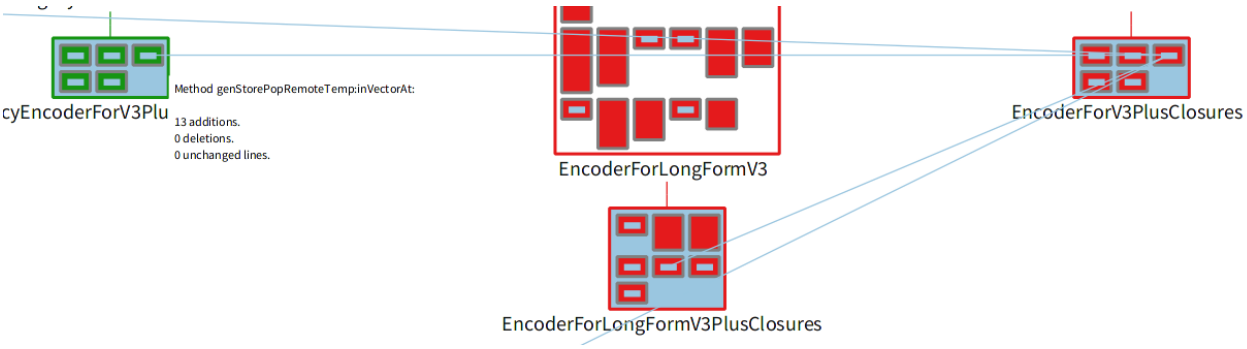


Figure 4.15: Method description tool-tip on the left, with similar methods highlighted. This is in *pharo-project/pharo* pull request number 166.

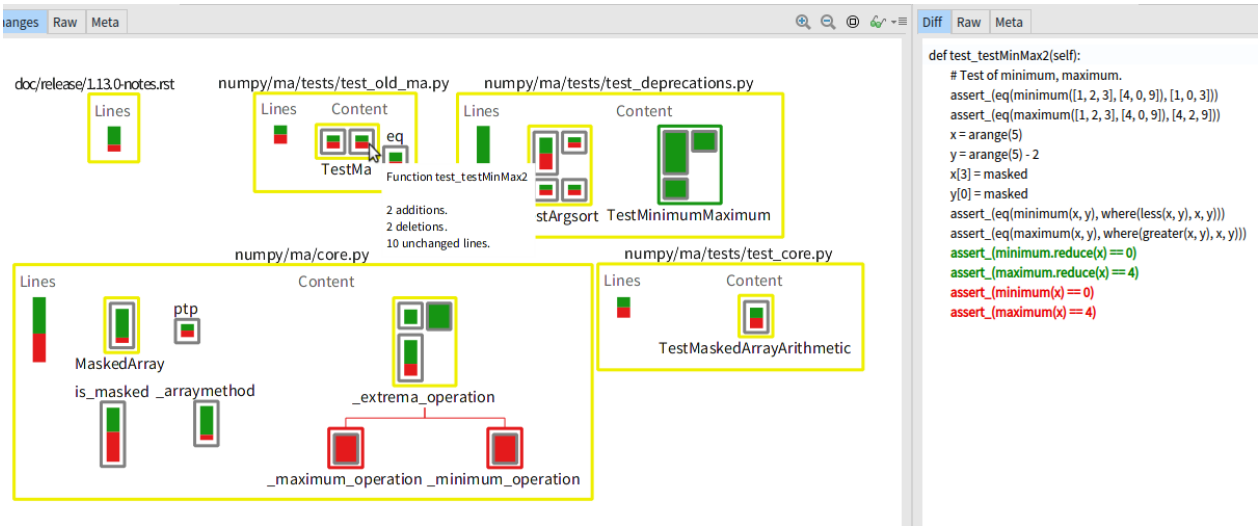


Figure 4.16: The textual diff of a Python method in *numpy/numpy* pull request number 8939.

4.7 Class changes visualization

The class changes visualization in Figure 4.18 is the same visualization described in Section 4.6, but without the packages and/or files boundaries separating the classes. The lack of these boundaries allows for a better display of the class hierarchies relationships, and how they are changing in the change set.

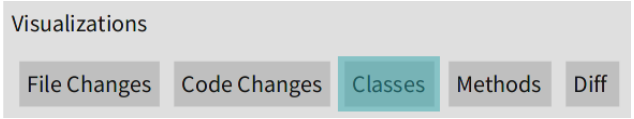


Figure 4.17: Classes accessing button

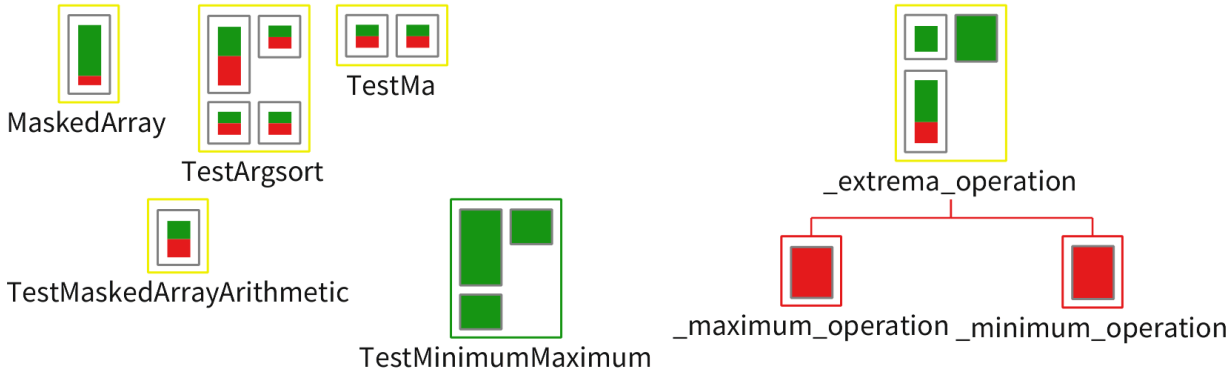


Figure 4.18: Class changes visualization for *numpy/numpy* pull request number 8939.

4.8 Method/function changes visualization

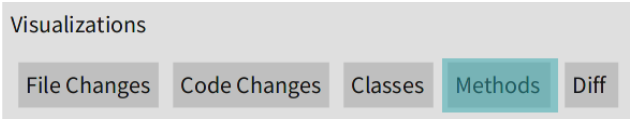


Figure 4.19: Methods accessing button

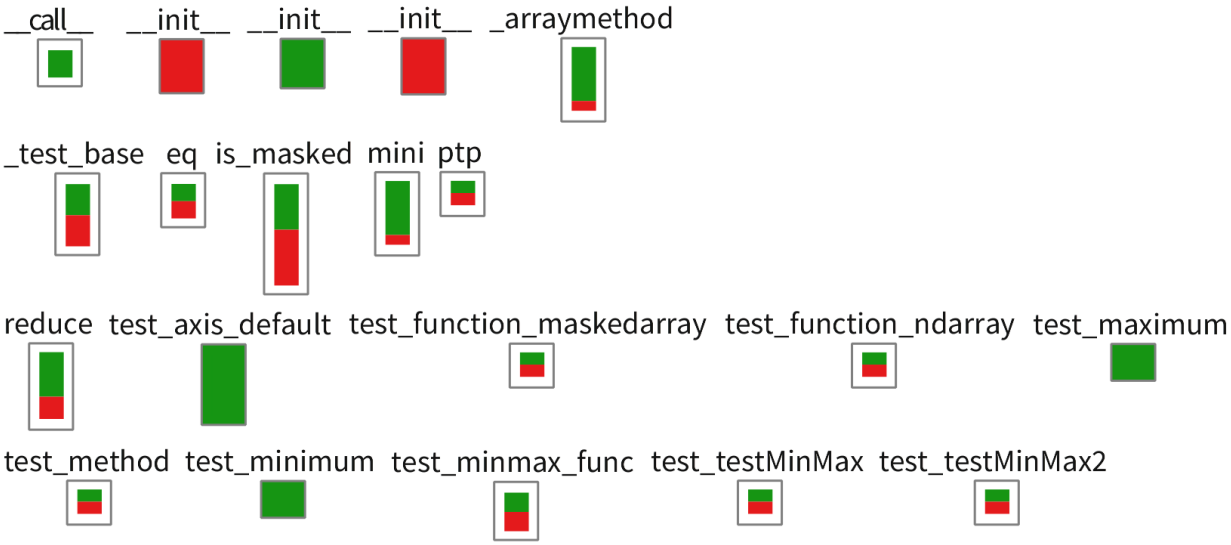


Figure 4.20: Method visualization for *numpy/numpy* pull request number 8939.

The method changes visualization in Figure 4.20 is for all of the methods and functions in the change set. This visualizations omits the packages and classes boundaries between methods. The methods are represented in the same way as in Section 4.6: a fully colored rectangle for completely added (green) or removed (red) methods; a rectangle with an embedded chart describing how the method is modified in terms of lines of code added (green), removed (red) or that are unchanged (light-gray).

4.9 Global Textual diff

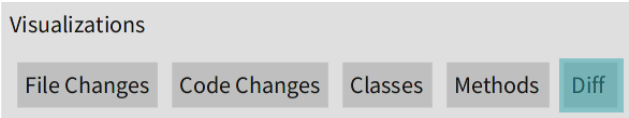


Figure 4.21: Global textual diff accessing button

The global textual diff view in Figure 4.22 is simply a view with the full textual diff with highlighting colors. In the case of *git* commits, this view simply presents the result of running the *git diff* command through a syntax highlighter so that added lines are in green, and removed lines are in red.

```
diff --git a/doc/release/1.13.0-notes.rst b/doc/release/1.13.0-notes.rst
index 2f32ddb28..3deca6a8c 100644
--- a/doc/release/1.13.0-notes.rst
+++ b/doc/release/1.13.0-notes.rst
@@ -28,10 +28,17 @@ Deprecations
 * Use of the C-API ``NPY_CHAR`` type number deprecated since version 1.7 will
   now raise deprecation warnings at runtime. Extensions built with older f2py
   versions need to be recompiled to remove the warning.
-* ``np.ma.argsort`` should be called with an explicit `axis` argument when
- applied to arrays with more than 2 dimensions, as the default value of
- this argument (``None``) is inconsistent with the rest of numpy (``-1``).
-
+* ``np.ma.argsort``, ``np.ma.minimum.reduce``, and ``np.ma.maximum.reduce``
+ should be called with an explicit `axis` argument when applied to arrays with
+ more than 2 dimensions, as the default value of this argument (``None``) is
+ inconsistent with the rest of numpy (``-1``, ``0``, and ``0``, respectively).
+* ``np.ma.MaskedArray.mini`` is deprecated, as it almost duplicates the
+ functionality of ``np.MaskedArray.min``. Exactly equivalent behaviour
+ can be obtained with ``np.ma.minimum.reduce``.
+* The single-argument form of ``np.ma.minimum`` and ``np.ma.maximum`` is
+ deprecated. ``np.maximum``. ``np.ma.minimum(x)`` should now be spelt
+ ``np.ma.minimum.reduce(x)``, which is consistent with how this would be done
+ with ``np.minimum``.

 Build System Changes
 ====================
diff --git a/numpy/ma/core.py b/numpy/ma/core.py
index 20cc77bc4..bccb0bce1 100644
--- a/numpy/ma/core.py
+++ b/numpy/ma/core.py
@@ -5621,6 +5621,14 @@ class MaskedArray(ndarray):
     """
     Return the array minimum along the specified axis.

+    .. deprecated:: 1.13.0
+      This function is identical to both:
+
+      * ``self.min(keepdims=True, axis=axis).squeeze(axis=axis)``
+      * ``np.ma.minimum.reduce(self, axis=axis)``
+
+      Typically though, ``self.min(axis=axis)`` is sufficient.
+
```

Figure 4.22: Global diff visualization

Chapter 5

Git Thermite design evolution

This chapter describes the different iterations and versions in the design of the Git Thermite tool. Each one of these versions went through at least a preliminary round of evaluation with at least a pilot version of a controlled experiment. These evaluation attempts were conducted with the researcher personally monitoring each one of the participants, so important tool design problems are detected by direct observation. The detection of these problems is what triggers going back to the drawing board and improving the design with the feedback obtained from the participants and through direct observation. The test subjects from pilot experiments, or from previous attempts on controlled experiments are not reused in the main controlled experiments of future iterations of the Git Thermite tool. Participants can only be reused in pilot controlled experiments.

5.1 First version: an attempt to make something original

Business card metaphor. The first iteration of the visual tool design is based around a single business card metaphor for GitHub pull requests. This is called the business card metaphor in analogy to a real world business card that someone gives to another as a quick introduction, to answer simple questions such as: *who are you you? what do you do? how can we contact you?*. The business card in our visualization is a card summarizing a change set. Some questions that should be answered by the business card for one change set are: *what are the changes? where are the changes?*(See Figure 5.2 for the file name tooltips) *what is the purpose of the changes?* (See the title in Figure 5.1 part 1) *how complex are the changes?*

In Figure 5.1 there is an example of the business card visualization for a change set. This business card is composed of the following three parts:

1. Title bar. The title bar has the pull request title; the type of pull request which is inferred with an heuristics (*e.g., Bug fix*) in Figure 5.1 part 1; and a colored square to indicate the possibility of doing an automatic merge of the pull request (Gray in Figure 5.1 part 1).

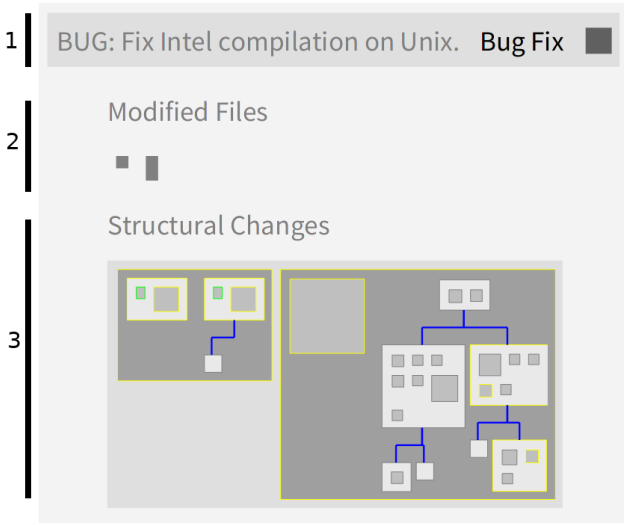


Figure 5.1: Business card visualization of the first iteration in the design of Git Thermite for *numpy/numpy* pull request number 9285. Part 1 is the title bar, part 2 is a *polymetric view* with the modified files, and part 3 is another *polymetric view* with the source code structure.

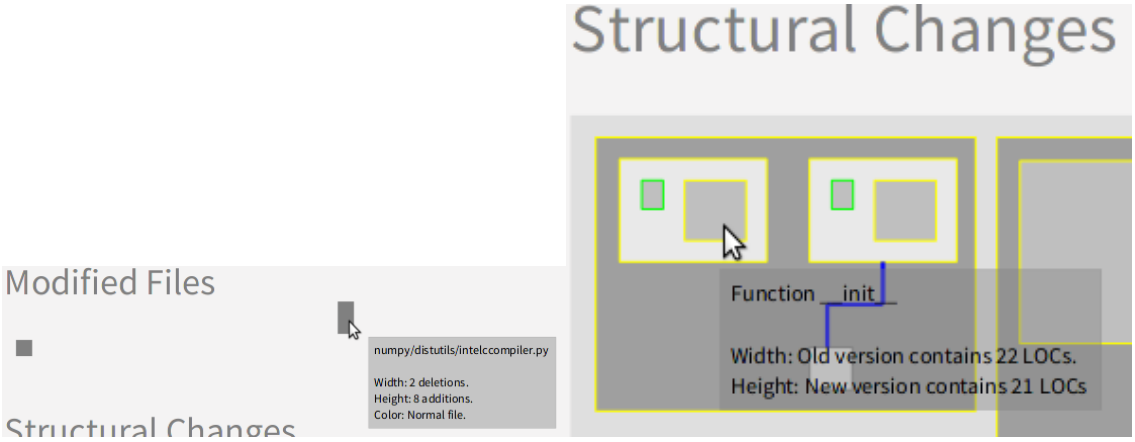
2. Modified files visualization. This is a *polymetric view* (See Section 2.4.1) where each modified file is represented with a rectangle. The width of these rectangles is mapped to the number of lines that are removed from the file, and the height is mapped to the number of lines that are added in the file. For example, in Figure 5.1 part 2 there are two modified files.
3. Structural code changes visualization. This visualization represents packages (Pharo) or files (Python) with boxes. Each package or file contains boxes with the classes present in the package or file. These classes can contain methods which are represented with boxes inside of each class. Class hierarchies are represented by lines connecting the classes. For example, in Figure 5.1 part 3 there are two Python files, with one class hierarchy in each of them. We decided to only display class hierarchies because they are easy and intuitive to understand.

Design concepts. The visualizations used during this version of the tool design are a direct application of the *polymetric views* by Lanza *et al.* [1] (See Section 2.4.1 for details). In the visualization for modified files, metrics are mapped as follow:

- Width: the number of deleted lines.
- Height: the number of added lines.
- Color: green if the filename contains the word *test*, otherwise it is colored gray. This is to emphasize the possibility of including unit tests.

In the case of the structural code changes visualization, the mapping of metrics is as follows:

- Width: the old number of lines of codes (LOCs).
- Height: the new number of lines of codes (LOCs).



(a) Tooltip in the modified files visualization. (b) Tooltip in the structural changes visualization.

Figure 5.2: Highlighting an element in a visualization with the cursor displays one of these tooltips.

- Border color: *green* for added element; *red* for removed element; *yellow* for modified element; and *gray* for unchanged element.

We choose these two mappings of metrics because it is a direct way to apply a *polymetric view* to the visualization of change sets. We also decided to use this kind of *polymetric view* because it is a standard visualization used for source code structural comprehension tasks.

Interactions. By hovering the boxes used in the visualizations with the mouse, a tooltip describing the visual elements used by the box is displayed (See Figure 5.2 for an example). For example, hovering a method displays a tooltip that explains the meaning of the width, the height and its border color, along with the actual number of lines of code in the old version, and the new version. By clicking on a method, class, package or file, the textual diff for the element is obtained in a separate window. This is in line with the idea of having an *omnipresent textual diff* suggested by Torch [4, 5, 21] (See Section 2.5.1 for an explanation of this concept). This implementation of the textual diff is retained until the final version of the tool that is described in Chapter 4.

Commit tree visualization. By clicking in the business card itself, a *commit tree* visualization for the pull request is displayed. This commit tree is a visualization for the history of a set of commits. In the case of Figure 5.3, there are three commits. The top-level commit on Figure 5.3 is the most recent commit. This top commit has two ancestor, one direct ancestor in the middle of Figure 5.3, and an older indirect ancestor on the bottom of Figure 5.3. Lines joining commits in this visualization indicate the relationship between one commits and its immediate ancestor. Only *merge* commits can have more than one ancestor, and they are the commits that produce a tree like appearance in this visualization. The commit tree visualization for a pull request tends to be a linear history as can be seen in Figure 5.3 most of the time because merge commits are rare in pull requests. The number of lines that

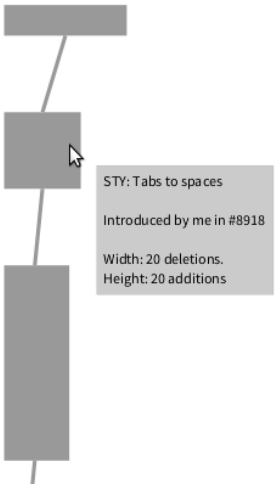


Figure 5.3: Commit tree visualization for *numpy/numpy* pull request number 8939 (Only first three levels). The vertical axis represents time, and each one of the boxes represents a commit in the history of the project. This visualization is an attempt to display the history of the change sets, however in its current state it does not provide helpful information.

are added in a commit is mapped to the height of the rectangles, and the number of lines that are removed in a commit are mapped to the width of the rectangles. We implemented the commit tree visualization in the tool with the intention of helping the analysis of pull requests. During the experimentation phases of the iterative tool design, the commit tree visualization was not tested and it did not provide useful information for the tested change set. For this reason, we removed the commit tree visualization for the the final tool design.

Pilot experiment. This version of the tool was tested in a pilot experiment with two Python industrial practitioners. This pilot experiment is a completely different experiment to the Python controlled experiment that was attempted with the final version of our tool, and it is presented in Appendix A. This pilot experiment involved analyzing three pull requests from open source Python projects, and one pull request from a *familiar project*[1] if possible. Conducting the experiment with familiar code was not possible in this pilot experiment because these practitioners do not use pull requests for their internal projects. These two pilot experiments were conducted in parallel, with the two participants in a single room. The experimenter was supervising the complete duration of this pilot experiment. During this pilot experiment, through direct observation of the participants it became clear that using length (width and height) to display source code changes is not intuitive, and it can even lead to confusion. One of the participants of this pilot experiment even suggested the idea of embedding charts to display the amount of changes. Since Torch [4, 5, 21] does something similar, this kind of feature is included in the next tool design iteration.

[1]Familiar project: a project that is proposed by the participant during the experiment. If the participant does not have a project to propose, then this step is omitted.

5.2 Second version: adding elements from Torch

Design improvements. The second iteration of the tool design builds incrementally on the design from the first iteration. The main change in this iteration is that source code changes, in terms of addition and deletions are not displayed anymore by using the width and height of the boxes that represent an element. Instead, the changes in terms of number of lines are displayed by using embedded bar charts in each one of the boxes that represent a modified element. These embedded bar charts represent the total number of lines added in green, the number of lines removed in red, and the number of lines that are unchanged with light gray in a single element.

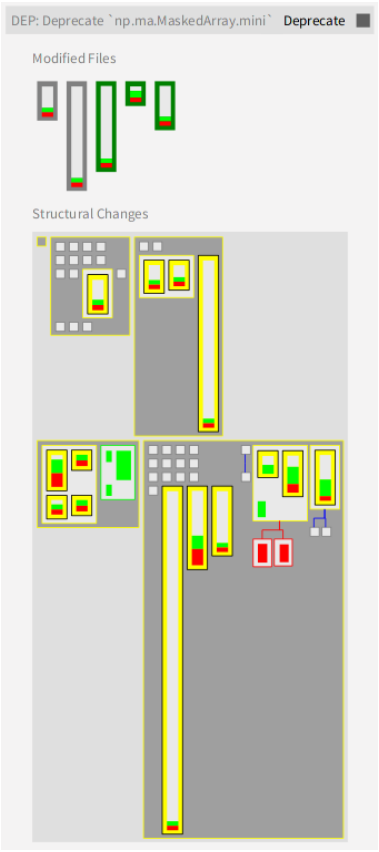


Figure 5.4: Second tool design iteration business card visualization for *numpy/numpy* pull request number 8939.

The use of embedded bar charts is similar to the colored bars that are used by Torch [4, 5] (See Section 2.5.1), however these charts are simpler because they only display totals. The charts used by Torch are richer because they represent how every line in a method is affected by a change set in terms of being a line added, removed or unchanged. We decided to not replicate this full colored bard visualization per method from Torch for simplicity reasons, and to also avoid potential cluttering in the visualization.

We also applied the same improvement of not representing changes by using dimensions (width and height) to the *commit tree* visualization, and instead use these embedded colored bars per commit. Like the case of methods, in the case of commits in the commit tree

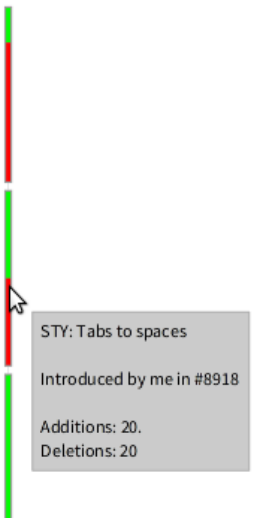


Figure 5.5: New version of the commit tree visualization *numpy/numpy* pull request number 8939 (Only first three levels). This version uses the new way for representing added and removed lines. Compare this version of the visualization with the previous one in Figure 5.3

visualization we are now representing the total number of lines added in a commit with a green rectangle, and the total number of removed lines in a commit with a red rectangle. For an example of this new version of the commit tree visualization, see Figure 5.5 and compare with the previous version of the same visualization in Figure 5.3.

Pilot experiment. For the second iteration of the tool design, a pilot experiment was conducted with two Pharo practitioners, regular contributors to open source projects. During this experiment, we received a positive user feedback for the visual tool from the participants. Due to these favorable results, we decided to attempt a controlled experiment with this version of the tool. This controlled experiment attempt is different to the controlled experiment for the final version of the Git Thermite tool that is presented in Chapter 7.

Controlled experiment attempt. An actual controlled experiment was attempted with four Pharo practitioners, two integrators and two developers from the core Pharo development team in France. This experiment consisted in analyzing three pull requests selected by the experimenter, and one pull request or commit selected by the participant.

Problems during the experimentation. Most of the pull requests used by this controlled experiment are composed mostly of big refactoring. Since the visual tool is only able to display added or removed elements, and it is not able to display moved or renamed elements, the visualizations produced by these pull requests tended to be very large. The visual tool tended to overestimate the complexity of the changes because of this problem. During this experiment the participant tried to guess refactoring by comparing the similarity in the visualization of elements added with elements removed. Despite being able to perform this visual comparison, the participants are not able to be sure whether something has been

moved or not by using just the visualization. Due to these problems we decided to consider this data as another pilot experiment for the evaluation of Git Thermite. To avoid introducing biases, we did not reuse these participants in the Pharo controlled experiment that is presented in Chapter 7.

Feedback from participants. Another important feedback received from an integrator is that they have a need to actually look at the whole code that is introduced in a pull request. One motivation for having this global textual diff is for finding snippets with malicious code, or source code comments with offensive expressions. They need to read the whole contribution in order to enforce some censorship, and community guidelines.

5.3 Third version: multiples visualizations

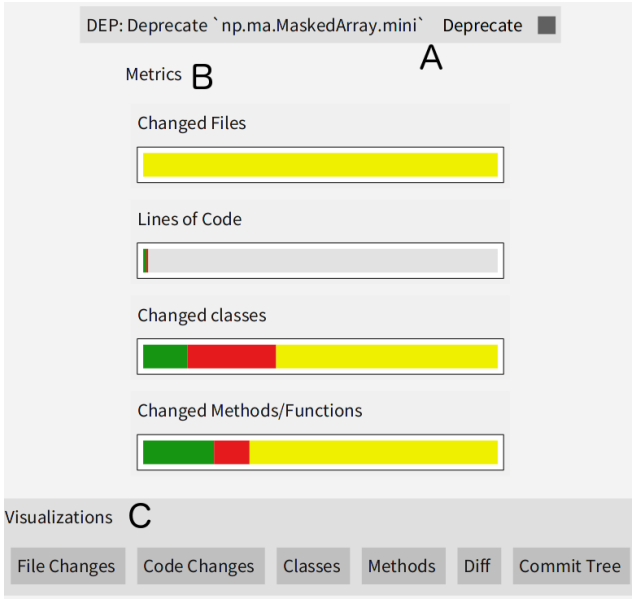


Figure 5.6: Third tool design iteration business card visualization for *numpy/numpy* pull request number 8939. Part A is the title bar of the business card. Part B is the section with metrics. Part C is the section with navigation buttons. Notice that the colored bar with the number of lines of code is dominated by the gray colored bar representing unmodified lines which makes it difficult differentiate the size between green (added lines) and red colored bars (removed lines).

Splitting the visualization. The third iteration of the tool design involves many changes. The most important change is the separation of the business card visualization, from the structural changes visualization. This separation is made with the intention of keeping the *business card* metaphor in terms of a card that gives quick information at a glance, and if the users wants more information they can then use the card for retrieving this information.

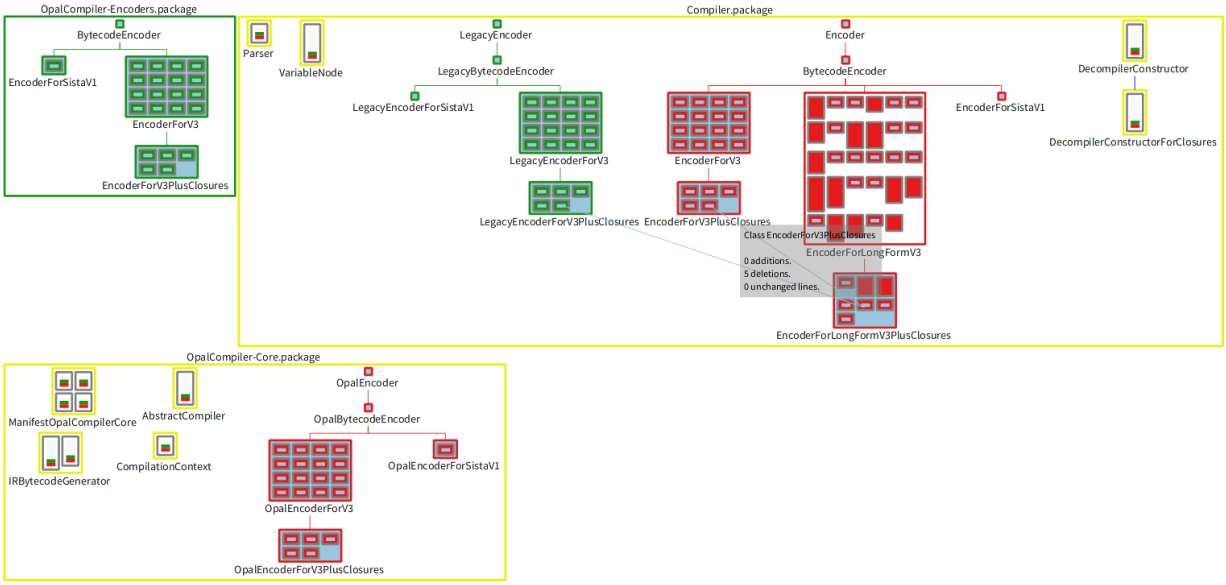


Figure 5.7: Third tool design iteration code changes visualization for *pharo-project/pharo* pull request number 166.

New business card visualization. The new business card for the change set can be seen in Figure 5.6. This new business card is composed of three parts:

- Title bar (Figure 5.6, part A). This part is the same as in the previous iterations.
- Metrics about the change set (Figure 5.6, part B). This part contains charts with source code metrics about the change set.
- Navigation buttons (Figure 5.6, part C). The final section is composed of buttons that allow the user to obtain additional visualizations about the change set.

In terms of interactions, the business card is a navigation hub for retrieving and building more detailed visualizations. The navigation buttons allow for visualizing different aspects of the whole change set: a visualization with all the packages; another visualization with all the classes, but ignoring the packages; or a visualization with only the methods. These interactions are also present in the final version, so for more details refer to Chapter 4.

Modified structural code changes. The visualization for structural code changes is moved into a separate view from the business card in comparison with the previous iteration. The saturation of the color scheme used by this visualization is reduced in order to make it more pleasant to the eye. However, the different hues of color retain their meaning: green for addition, red for deletions, yellow for modifications, grey for unchanged elements, and cyan for moved elements. Classes and methods that are added or removed that have the same source code are represented with a cyan color to indicate code movement. Edges connecting elements with the same source code can be displayed by highlighting one cyan element with the cursor (*e.g.,* Figure 5.7).

Pilot experiment. The pilot experiment in this tool design iteration was performed with one Pharo practitioner from the academia, and one Python practitioner also from the academia. This pilot experiment shows some small issues that are fixed for the final tool iteration. One issue is related to the lack of some reference points in the code metrics charts that give a sense of scale to the metrics displayed. Another problem is related to some issues in the unmodified code filter, which makes the lines of code chart chart useless for large software as can be seen in Figure 5.6 where the white colored bar representing the *unmodified lines of code* are dominating the *green* (added lines) and *red* (removed lines) *bars* whose size looks almost the same. Finally, the existence of an untested *commit tree* visualization (See Figure 5.3 and Figure 5.5) is just adding noise to the tool and the experiments.

5.4 Final version: consolidating the tool design

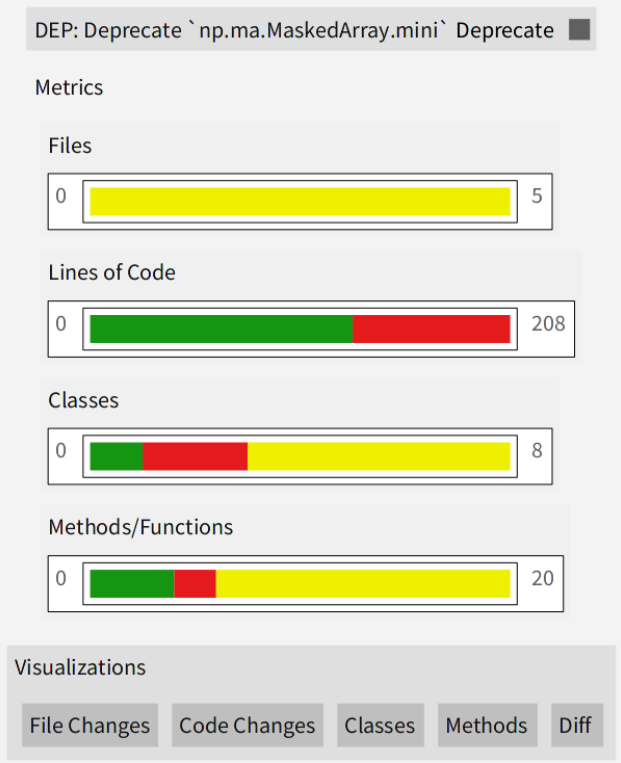


Figure 5.8: Final tool design iteration business card visualization for *numpy/numpy* pull request number 9285.

The final iteration is an incremental improvement from the previous iteration. For a complete detailed description of the visual tool used in this iteration, refer to Chapter 4. The changes introduced during this tool design iteration are the followings:

- The addition of numbers to the left and right side of metrics charts. In order to improve the sense of scale, the bar plots in the business card visualization now have a 0 number on the left side, and the total number of elements represented by the colored bars on the right (See Figure 5.6 in comparison with Figure 5.8).

- Removing the commit tree visualization (See Figure 5.3 and Figure 5.5). For the purposes of the experimental studies that are performed for this thesis, the *commit tree* visualization does not provide useful information. Most of the experiments are analyzing just a single commit, so this visualization is composed with only a single rectangle without providing additional information to the user.
- Adding charts with the number of lines of code that are added, removed or are unchanged to the left side of the package or files in the structural changes visualization (See Figure 5.6 in comparison with Figure 4.14).

Pilot experiment. The pilot experiment for the final iteration of the tool design is conducted with two Pharo practitioners. This pilot consisted of analyzing four moderate to large change sets of an unfamiliar code project. Two of these change sets are analyzed using the visual tool, and two commits are analyzed using Monticello textual diff tool as the baseline tool. The two participants analyzed the same commits, but not with the same tool in order to be able compare them.

This pilot experiment presented only few problems that were corrected for the final experiment. For details on these problems, the controlled experiment design, and the final experiment execution, refer to Chapter 6.

Chapter 6

Controlled experiment design

In software engineering, there are five classes of empirical research methods [27]: Controlled Experiments; Case Studies; Survey Research; Ethnographies; Action Research. A *case study* is a qualitative empirical method that can be *exploratory* and *confirmatory* [27]. *Survey research* consists in polling members of a community with a questionnaire with the objective of answering a clearly defined research question [27]. An *ethnography* is a sociological study of a community through the direct observation of a researcher that gets immersed in the community [27]. Action Research is an empirical method where "researchers attempt to solve a real-world problem while simultaneously studying the experience of solving the problem" [27]. A controlled experiment is a quantitative empirical method for testing a hypothesis in a laboratory setting. In a controlled experiment *independent variables* are manipulated to measure their effect in *dependent variables* [27].

We performed two controlled experiments in this thesis to evaluate Git Thermite: one controlled experiment with Pharo practitioners (See Chapter 7); and one controlled experiment with Python practitioners (See Appendix A). We chose to use controlled experiments because they allow us to measure the relationship between independent variables, and dependent variables. In the case of this thesis, the main independent variable is the *tool* used to analyze the change sets. The two main dependent variables are the *time* taken to analyze the commit, and the *precision* of the analysis results. The objective of the controlled experiment is to find the relationship between these variables in order to test a hypothesis.

Unfortunately, the execution of the Python experiment displayed an important bug in the visual tool implementation related to the parsing of Python code. Constructing, an appropriate Python parser for Git Thermite is beyond the scope of this thesis. Due of this problem, and the difficulty of finding Python developers with some experience in reviewing source code, we decided to not further pursue the experimentation in Python. However, for sake of completeness, the description of this experiment attempt is still included in this thesis in Appendix A.

6.1 Similar controlled experiments

We are designing our controlled experiment by basing on other works in the literature that perform a controlled experiment for validating a software visualization: CodeCity [2, 22] and CuboidMatrix [29]. CodeCity [2, 22] is a software visualization that uses a city metaphor to represent software. CuboidMatrix [29] is another software visualization that builds upon the space-time cube metaphor that was originally created in the field of data visualization [23]. Of particular importance is the work by CodeCity as it involves a detailed explanation on its experimental design and execution [2], which we use as guide for our experiment. The controlled experiment is divided in three main phases:

1. Controlled experiment design.
2. Pilot experiment.
3. Experiment execution.

Experiment design elements. The design of a controlled experiment involves the following steps:

1. An experimental design wish list [2].
2. The selection of research questions [2] along with corresponding *null hypotheses* and *alternate hypotheses* to be tested during the experiment.
3. The identification of dependent and independent variables.
4. The identification of factors that can affect the outcome of the experiment. These factors are variables that need to be controlled [2].
5. The definition of tasks that the participant has to perform during the controlled experiment [2].

Experiment design wish list. We selected the following desirable traits from the CodeCity experiment design wish list [2]:

1. Choose a fair baseline for comparison [2]. Choosing a fair baseline can be complicated. In our case, we chose as baseline the *git diff* command line tool for analyzing the differences between arbitrary git commits; in the case of Pharo, our baseline is the Monticello diff tool (See Section 2.1.3), because it is the tool normally used to compare changes in Pharo.
2. Involve participants from industry [2]. It is preferable to involve participants from the industry because the objective of the tool is to support them [2]. However, because the Pharo programming environment is mostly used in research, getting Pharo practitioners from the industry can be complicated.
3. Provide a tutorial of the experimental tool to the participants [2].
4. Find a set of relevant tasks [2].
5. Include tasks which may not take advantage the tool being evaluated [2].
6. Choose real-world systems [2].

7. Provide the same data to all participants [2].
8. Report results on individual tasks [2]. See Section 7.4 and Section A.5.

6.2 Research questions and hypotheses

The research hypothesis of this thesis as described in Section 1.3 is the following:

By using a visual analysis and reporting tool that displays metrics, the changes to the structure of the software and the source code, in addition to the information presented by the textual diff, it is possible to facilitate code reviewing process in terms of time, comprehension of the changes and cognitive load to the reviewers.

By distilling this hypothesis, we have the following research questions to be answered by the experiment:

- Q1: Is the analysis of source code change set *faster* by using Git Thermite instead of a textual diff tool[1].
- Q2: Do the source code analysis that are done with Git Thermite have more *precision* than the analysis that are done with a textual diff tool?
- Q3: Does Git Thermite allows inferring more information about the *intention of changes* than textual diff?
- Q4: Does Git Thermite reduce the cognitive load (*i.e.,* reduce the mental stress) for analyzing a source code change set in contrast with a plain *textual diff*?

For each one of these questions we formulate the corresponding null hypotheses[2] and alternate hypotheses[3] in Table 6.1.

6.3 Controlled experiment tasks and questions

The Git Thermite controlled experiment is composed of the following tasks:

1. Questions about the participant himself.
2. Git Thermite learning material reading, and tool practice.
3. Unfamiliar code analysis.
4. Familiar code analysis.
5. Questions about the user general impressions on the tool.

[1] In this context, textual diff tool also applies to the Monticello changes tool.

[2] A null hypothesis represents a general statement or default position about the relationship between two measured phenomena(*e.g.,* no relationship). The term null hypothesis comes from statistical hypothesis testing.

[3] An alternative hypothesis is any hypothesis that is formulated with the purpose of contrasting and testing against the null hypothesis.

	Null hypothesis	Alternate hypothesis
H1	There is no *time difference* between using *Git Thermite* and using a *textual diff tool.*	Commit analysis is significantly *faster* by using *Git Thermite* than by using a *textual diff tool.*
H2	There is no difference in terms of *analysis precision* between using *Git Thermite* and using a *textual diff tool.*	The source code analysis that are made by using *Git Thermite* have significantly more *precision* than the analysis that are made by using a *textual diff tool.*
H3	The intention of changes inferred by using *Git Thermite* is the same intention inferred by using a *textual diff tool tool.*	The intention of changes inferred by using *Git Thermite* is more detailed than the intention inferred by using a *textual diff tool.*
H4	The cognitive load of doing an analysis by using *Git Thermite* is the same as doing the analysis by using a *textual diff tool.*	There is less measurable cognitive load by using *Git Thermite* instead of using a *textual diff tool.*

Table 6.1: Null and alternate hypotheses.

6. Filling the NASA TLX form.

The questions used for these different sections are in Section 7.1 for the Pharo experiment, and in Section A.1 for the Python experiment. Since the Python experiment was performed after the experiment with Pharo practitioners, we had the opportunity to improve the questions for the Python experiment. For the Python experiment, we changed the questions in order to improve their clarity, and to facilitate the posterior analysis.

Questions about the participants. The first task given to the participants of the experiment consisted in answering some personal questions. The questions in this task are for gathering information about the developer experience, and familiarity with the experiment elements: programming language, usage of git, software visualization, and the familiar source code to be analyzed. The questions applied during the Pharo controlled experiment are present in Section 7.1.1. As for the questions applied during the Python controlled experiment attempt, they are present in Section A.1.1.

Git Thermite learning material reading, and tool practice. In the second task we asked every participant to read a description of the Git Thermite tool for learning purposes. A copy of the exact learning material used in this task is provided in Appendix B.

Unfamiliar code analysis. This task consisted of answering questions about an specific change set by using either Git Thermite, or a textual diff tool. The questions asked in this task are about characterizing the change set, identifying the presence of unit tests, and trying to identify the intention of the changes. The questions applied during the Pharo controlled

experiment are present in Section 7.1.2. As for the questions applied during the Python controlled experiment attempt, they are present in Section A.1.2. We selected the change sets, and the tools used in this task before the experiment started. The selection of commits for this task is made in a global basis. For this task, we applied the same commits to all of the participants with the objective of having comparable results for different participants. However, for each participant we randomized the order used by them to analyze the commits with the objective of reducing *learning bias*[4]. Each commit is only analyzed one time by a participant. The assignments of change sets, tools for doing a change set analysis, and the order of analysis for each participant of the Pharo controlled experiment is described in Section 7.3 and in Table 7.5. In the case of the Python controlled experiment attempt, these assignments are described in Section A.3 and in Table A.7. The order for using Git Thermite and the baseline tool is interleaved (*e.g.,* first one tool, then the other tool, then back to the first, *etc.*).

Familiar code analysis. The task of analyzing familiar code uses exactly the same questions as the task for analyzing unfamiliar code. The only difference in this task is that the application to be analyzed by the participant is suggested by the same participant. This code is called *familiar code* because the participants knows the source code from before the experiment (*i.e.,* the participant has experience working with the source code). In case the participant is not capable of providing a project with familiar code for the session, this task can be completely omitted. Since this task is completely dependant on each participant, the selection of commits for this task is made at the beginning of the session by the experimenter once the participant provides to a project repository (usually a private repository) to the experiment. The criteria of selecting commits for this stage is looking for non-trivial commits that generate visualizations or textual with a moderate to large size (*e.g.,* multiple classes and methods). Since most of the commits in the history of these repositories tended to be very simply or trivial, most of the time we choose simply intervals composed of several commits back in the history of a project. These intervals included between 10 and 50 commits. We did the selection of these commits during the time the participants were answering the personal questions of the first task on paper or in a separate machine. Since the commits for this task are selected per participants, the results between different participants for this task are not directly comparable between different participants.

General impressions. After the code analysis task, the participant is asked to answer some questions about his subjective impressions of the tool, and to give some general feedback.

NASA TLX form filling. The NASA Task Load Index (TLX) [28] is a form with a standard scale for measuring cognitive load when performing a task. Since this form and methodology is widely used [28] in a variety of fields, we just reuse it as a part of our experiment. At the end of the experiment, we apply two copies of this form to the user,

[4]As participants are analyzing different different commits, they get better on analyzing the commits because they also are learning on how to use the tool. This learning process introduces a bias in the experiments result.

one copy to be filled about the user experience with Git Thermite, and another copy for the user experience with the *textual diff tool* used during the experiment. The NASA TLX forms ask the participants to answer the following questions with an integer between 1 and 21, inclusive:

- *Mental Demand*: How mentally demanding was the task? (1 is very low, 21 is very high)
- *Physical Demand*: How physically demanding was the task? (1 is very low, 21 is very high)
- *Temporal Demand*: How hurried or rushed was the pace of the task? (1 is very low, 21 is very high)
- *Performance*: How successful were you in accomplishing what you were asked to do? (1 is perfect, 21 is failure)
- *Effort*: How hard did you have to work to accomplish your level of performance? (1 is very low, 21 is very high)
- *Frustration*: How insecure, discouraged, irritated, stressed, or annoyed were you? (1 is very low, 21 is very high)

Chapter 7

Pharo controlled experiment

This chapter describes the controlled experiment with Pharo participants. The questions applied to the participants are described in Section 7.1. The selection of commits to be analyzed during the the *Unfamiliar code analysis* task is described along with the execution of a *Pilot experiment* in Section 7.2. The participants and assignments of change sets to analyze is described in Section 7.3. The results of this controlled experiment with a minimal analysis is presented in Section 7.4. A further detailed discussion and analysis of these results is presented in Chapter 9.

7.1 Questions

This section contains the questions that we applied to the participants of the Pharo controlled experiment. For reasons of accuracy, we are reproducing the English mistakes that we commited on redacting these questions. For example, "Characterize the size and complexity of the commits?" is not even phrased as questions, so the question mark should not be present. If the experiment is replicated, these English mistakes should be fixed.

7.1.1 Personal questions

The questions used in the first task in the Pharo experiment are the followings:

1. How many years of experience do you have in programming?
2. How long have you been programming in Python?
3. How long have you been programming in Pharo?
4. Which other languages do you use?
5. How old are you?
6. What is your gender?
7. What is your highest achieved academic degree?

8. Are you a student or a professional practitioner? (Two alternatives)
9. How familiar are you with using visualizations? Choose one of the following:
    - Zero: I have never used a visualization.
    - Low: I almost never use a visualization.
    - Medium: I regularly use a visualization.
    - High: I use quite frequently a visualization, or it is part of my daily job.
10. Do you use Git? (Yes, or no)
11. Which other version control systems do you use?
12. Have you made a commit in GitHub? If yes, do you use or make commits on GitHub on a regular basis?
13. How do you review source code changes?
14. How do you integrate commits made by a developer other than you?
15. How do you verify a commit made by a developer other than you?
16. How do you navigate in the history of a project?
17. What are the limitations you see in Monticello and/or Git diff?
18. Do you consider yourself an expert in the application used in this study?
19. Who do you think are the experts in the application used in this study?

Most of these questions are open questions (*i.e.,* without alternatives to choose) given to the participant. The reason to select open questions is to extract as much information as possible from the participants of the experiment.

7.1.2 Source code change set questions

The following is the list of questions used in the *source code change set analysis* tasks, in the Pharo experiment:

1. Characterize the size and complexity of the commits? (trivial vs complicated) Please, justify.
2. Do the commits actually include unit tests? Please, justify.
3. How are the changes in the commits? Are things being mostly added, removed, modified, or something else? Please, justify.
4. Is there a refactoring (class rename, code movement) in the commits? Please, justify.
5. Can you describe the class hierarchy involved in the pull request / commit? Please, justify.
6. Is easy to understand the changes in the commits? (strongly agree, agree, neutral, disagree, strongly disagree) Please, justify.

Since our original motivation is in the context of facilitating the integration of a *pull request*, we choose these questions because they are of interest to an integrator of an open source project (See Section 1.1). These questions are used for analyzing both, unfamiliar and familiar code. In addition to these questions there are also following extra data fields in the questionnaire form for analyzing a particular commit:

1. The tool used to analyze a particular change set.
2. The name of the change set analyzed.
3. The commit analysis starting time.
4. The commit analysis ending time.

The objective of the first question (*"Characterize the size and complexity of the commits? (trivial vs complicated)"*) is for the participant to write a general overview and a coarse grained description of the change set that is analyzed. Instead, the objective of the third question (*"How are the changes in the commits? Are things being mostly added, removed, modified, or something else?*) is for the participant to write a detailed description of the change set that is analyzed. All of these questions are asked as open questions where the participant has to write a justification with words.

7.1.3 Questions about the user general impressions on the tool

The following is the list of questions used for the task about the user *general impressions* in the Pharo experiment:

1. What do you think about the content visualization?
   - I find it easy to understand in general: (strongly agree, agree, neutral, disagree, strongly disagree) Please, justify.
   - I find it easy to understand the visual variable mapping: (strongly agree, agree, neutral, disagree, strongly disagree) Please, justify.
   - I find it easy to understand the interactions: (strongly agree, agree, neutral, disagree, strongly disagree) Please, justify.
   - I find it easy to navigate between visualizations: (strongly agree, agree, neutral, disagree, strongly disagree) Please, justify.
   - I find it useful for solving requested tasks: (strongly agree, agree, neutral, disagree, strongly disagree) Please, justify.
   - I find it useful for my daily job: (strongly agree, agree, neutral, disagree, strongly disagree) Please, justify
2. In what scenarios and for solving which maintenance tasks would developers use Git Thermite? Please, justify.
3. What tools would you use instead of Git Thermite to retrieve the same information? Please, justify.
4. Do you have a suggestion on how to improve the visual report or the tool itself? Any critique? Please, justify.

7.2 Pilot

The Pharo pilot experiment was conducted with two Pharo practitioners. For this experiment we had problems on selecting medium to large change sets in Pharo, so we chose to

use different ranges of commits in the recent history of Roassal2 [30]. Since our original motivation involves facilitating the integration of GitHub pull requests (See Section 1.1), we should have selected pull requests from open source Pharo projects on GitHub. However, the movement from Monticello based projects to GitHub on the Pharo community is recent, and they are few Pharo projects on GitHub with pull requests. For this reason, we chose to use these commits made in Monticello as a substitute for GitHub. This also implies, that we can use the standard Monticello diff tool (See Section 2.1.3) used by the Pharo community as a fair baseline tool in our experiment. Each participant in the experiment had to analyze four change sets from Roassal2: two change sets with Git Thermite; and two change sets with the Monticello diff tool. Table 7.1 displays the change sets from Roassal2 that we selected. The participants of this pilot experiment are not reused during the proper execution of the controlled experiment described in Section 7.3.

Change set number	Old Monticello commit	New Monticello commit
CS1	1619	1648
CS2	1597	1617
CS3	1572	1596
CS4	1534	1545.

Table 7.1: Pharo experiment change sets selected from Roassal2.

In Figure 7.1 we have the business card visualizations for the four different change sets that are analyzed during the *Unfamiliar code analysis* task of both, the *pilot experiment* and the proper run the the Pharo controlled experiment. The visualization for structural changes of CS1 and CS2 is present in Figure 7.2, and this visualization for CS3 and CS4 is present in Figure 7.3.

7.3 Execution

Experiment application. The application of the questionnaire was done on paper during the pilot experiment, and for the first two participant. To avoid having to manually transcribe the remaining participant answers during the posterior analysis, we moved the questionnaire into Google Form[1]. We kept the open nature of the questions in the digital form version. Since the NASA TLX form [28] requires making a mark with a pencil, we always applied this form on paper. The sessions for Participant 1 and 2 were conducted with the experimenter physically present on the same room as the participant. The sessions for the remaining of the participants were conducted remotely through a video conference. During these remote session, the experimenter was observing the screen of the participants, and giving instructions to the participants during the whole session. The experimental tool setup was done by: lending a laptop to the participants on the physical sessions; or providing a whole Pharo environment (the whole image) with a single file to download in the case of the remote sessions.

[1]Google Form is a platform for asking questions through a web page.

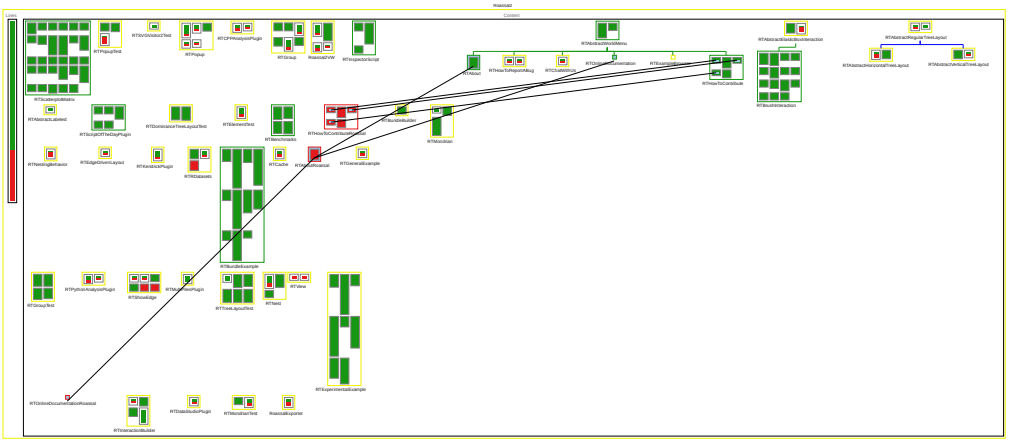
(a) CS1 (b) CS2

(c) CS3 (d) CS4

Figure 7.1: Business card visualizations for the different change sets.

Participants. The Pharo experiment was conducted with five Pharo practitioners. The Pharo practitioners have a profile that comes from both the academy and the industry. They have achieved at least an engineering degree and have an extensive programming experience of at least 7 years in general, and 5 years of programming in Pharo. For details in the participants background see Table 7.2. The main criteria for selecting these participants are the followings: the know how to use Pharo; they are not undergrad student (*i.e.*, the have some professional experience); and they were willing to participate on the controlled experiment.

Experience with visualizations, Git and GitHub All of the participants are reporting at least a medium level of familiarity with using visualizations in general. The participants reported that they use Git, and the Monticello version control system. All of the participants report that they have performed at least one commit in GitHub, but they do not report the same frequency of usage. For details on this reported data, see Table 7.3.



(a) CS1

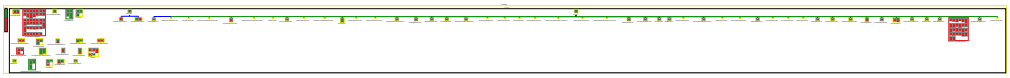
(b) CS2

Figure 7.2: Structural changes for CS1 and CS2.

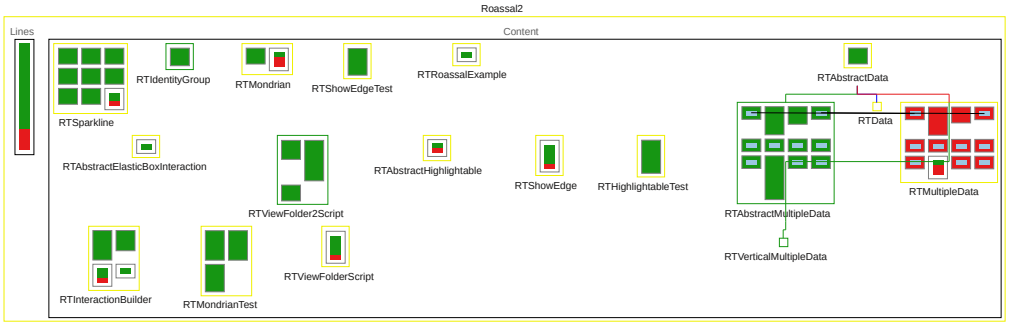
	Programming years of experience	Pharo programming experience	Highest academic degree	Student or professional?
Participant 1	7	5 years	Master	Professional
Participant 2	7	6 years	Engineer	Student, PhD
Participant 3	7	6 years	PhD	Professional
Participant 4	14	7 years	Master	Professional
Participant 5	7	7 years	Engineer	Professional

Table 7.2: Pharo experiment participants background.

Experience with source code reviewing. Some participants report doing source code reviewing occasionally, and the rest of the participants do not review the source code written by others. These participants clearly are not integrators of source code changes. Some participants report using the tools provided by collaborative platforms such as GitHub and GitLab for navigating in the source code history. One participant uses a custom desktop application for navigating in a project history (Participant 4: SourceTree.app), and another participant reports to almost never having to navigate in a project history (Participant 2). Notice that Participant 4 reports using GitHub on an almost daily (See Table 7.3) basis, but also reports that he or she does not have the experience of reviewing source code yet, and that he or she has used the "GitHub web page before, a few time" (See Table 7.4), the answers by this participant may seem inconsistent, but it is not because the participant is using GitHub only for commiting his or her own code or for making contributions to open source projects that are owned by others. Participant 4 does not review the source code of other developers because he or she does not an integration role in an open source project that receives pull request or external contributions. For details on the participants experience



(a) CS3



(b) CS4

Figure 7.3: Structural changes for CS3 and CS4.

	Familiarity with using visualizations	Do you use Git?	Which other VCS do you use?	Have you made a commit in GitHub?	Do you use or make commits on GitHub on a regular basis?
Participant 1	Medium	Yes	Monticello	Yes	A long time ago (3 years)
Participant 2	Medium	Yes	Monticello	Yes	Yes
Participant 3	Medium	Yes	Monticello	Yes	Two times a week
Participant 4	Medium	Yes	Monticello, SVN	Yes	I do it on a regular (almost daily) basis
Participant 5	High	Yes	Monticello	Yes	Yes

Table 7.3: The experience of Pharo participants with visualizations, Git and GitHub.

with source code reviewing see Table 7.4.

Change set assignment. For the Pharo experiment we used the same change sets that we used in the pilot experiment (See Section 7.2). Each one of the participants of this experiment analyzed the same four change sets (CS1, CS2, CS3, and CS4) during the *unfamiliar code analysis* task. However, the order of analysis of these change sets is randomized with the objective of reducing learning bias. The tool used for doing this analysis (Git Thermite or the baseline) is also randomized per change set. This randomization is performed before the experiments started, and the change sets and tool assignments produced by this randomization is presented in Table 7.5. Note that all participants are analyzing all the commits, and they are analyzing two commits with Git Thermite, and two other commits with baseline textual diff tool (Monticello diff in this case).

	How do you review source code changes?	How do you verify a commit made by a developer other than you?	How do you navigate in the history of a project?
Participant 1	Using Monticello changes or Git diff	Manual source inspection and tests	Using GitLab, Bitbucket, etc tools
Participant 2	With diff	I test it and look a bit the overall changes.	I almost never do this.
Participant 3	Yes	I use the change diff tool	I use Monticello browser
Participant 4	I do not review much as I am main contributor to my code. I do not read other's code regularly.	I do not have this experience yet. I used the GitHub web page before, a few times.	I use SourceTree.app
Participant 5	Historial of Git	Click on the commit and see all the changes	I use to choose certain files and see the changes

Table 7.4: Pharo participants experience with source code reviewing.

	First change set	Second change set	Third change set	Fourth change set
Participant 1	CS1 Thermite	CS2 Diff	CS3 Thermite	CS4 Diff
Participant 2	CS3 Diff	CS4 Thermite	CS1 Diff	CS2 Thermite
Participant 3	CS3 Thermite	CS4 Diff	CS1 Thermite	CS2 Diff
Participant 4	CS1 Diff	CS2 Thermite	CS3 Diff	CS4 Thermite
Participant 5	CS2 Thermite	CS4 Thermite	CS1 Diff	CS3 Diff

Table 7.5: Pharo experiment change set assignment

Problems during the execution. In one case (Participant 1), due to time constraints of the participant, it was not possible to conclude the experiment in a single session. In this case, the *familiar code analysis* task was performed in a separate session, but all of the rest of the experiment was concluded during the first session.

7.4 Results

This section contains the graded and summarized answers given by the participants of the Pharo experiment. For reasons of space, the exact answers given by the participants are not

included in this section. However, the exact answers given by the participants are available in Appendix C. The only modifications made to the answers in Appendix C are to ensure the anonymity of the participants, especially by hiding the name of the applications analyzed in the *familiar code analysis* task.

7.4.1 Unfamiliar code analysis

This section presents the results for the unfamiliar code analysis task of the Pharo controlled experiment. Since each one of the participants answered the same questions for the same four different change sets, but in a different order, and with a different tool, the results for different participants analyzing a same change set are comparable. For this reasons, we are organizing the different tables of this section by grouping the results per change set to facilitate a direct comparison of the results.

Size and complexity characterization. For the question: *Characterize the size and complexity of the commits (trivial vs complicated) Please, justify.* We decided to score according to the presence of elements in the answers given by the participants, such as whether the change set is complicated or not, the presence of numbers such as LOCs or number of affected classes or methods, or whether the participant uses an example in the answer to this question. For example, participant 1 answered the following for CS1:

"It is a large commit with more than 5000 lines changed. Also it is complex because there are too many changes of different domain. There should split the commit." Since the participant says this is a large commit, we score 1 for *"Is it complicated?"*. Since the participant also mentions "more than 5000 lines changed" we also score 1 for *"Presence of LOC?"*.

This grading for all of the change sets is given in Table 7.6. These results do not highlight a tendency that favors one tool or the another. As another example, participant 4, answered the following for CS2:

"It is a complicated set of changes as there are 1393 changed lines of code, affecting 50 classes and 203 methods."

In this case, the participant is also mentioning the number of classes, and the number of methods. We decided to use this scoring procedure for this question with the objective of simplifying the analysis of the gathered data, however, this analysis may be biased by the personality of the different participants.

	Tool	Is it complicated?	Presence of LOC?	Presence of class count	Presence of method count	Presence of examples and/or tests	Presence of example of changes	Score
Unfamiliar CS1								
Participant 1	Thermite	1	1	0	0	0	0	2
Participant 2	Diff	1	0	0	0	0	0	1
Participant 3	Thermite	1	0	0	0	0	1	2
Participant 4	Diff	1	0	0	0	0	1	2
Participant 5	Diff	1	0	0	0	0	0	1
Unfamiliar CS2								
Participant 1	Diff	0	0	0	0	0	1	1
Participant 2	Thermite	1	1	0	0	0	0	2
Participant 3	Diff	1	0	0	0	0	1	2
Participant 4	Thermite	1	1	1	1	0	0	4
Participant 5	Thermite	1	0	1	0	0	0	2
Unfamiliar CS3								
Participant 1	Thermite	0	0	0	0	1	0	1
Participant 2	Diff	1	0	0	0	0	0	1
Participant 3	Thermite	0	0	0	0	0	1	1
Participant 4	Diff	0	0	0	0	0	0	0
Participant 5	Diff	0	0	1	0	0	0	1
Unfamiliar CS4								
Participant 1	Diff	0	0	0	0	0	0	0
Participant 2	Thermite	1	1	0	0	0	0	2
Participant 3	Diff	0	0	0	0	0	0	0
Participant 4	Thermite	1	0	0	0	0	0	1
Participant 5	Thermite	0	0	0	0	0	0	0

Table 7.6: Unfamiliar Pharo code: grading of question *Characterize the size and complexity of the commits (trivial vs complicated)*

Presence of unit tests. As for the second question: *Do the commits actually include unit tests? Please, justify* All of the participants were correct in their answers. All of the participants properly identified whether or not there were unit tests for all of the change sets independent of the tool that they were using to analyze the change sets.

Description of changes. For grading the question: *How are the changes? Are things being mostly added, removed, modified, or something else? Please, justify.* We checked the presence of numerical values in the description given by the participants, and the visual variables used by the participants. Like the previous case, the presence of these two variables on the answers may be biased by the personality of the participants. In addition to extracting these elements from the descriptions given by the participants, we made a summary of the description given by the participants with the objective of trying to find consensus. For these results, see Table 7.7. As an example of this analysis and summarizing procedure, here are the textual answers given by the participants for CS1:

- Participant 1: Mostly added. 113 new methods vs 44 modified vs 6 removed.

- Participant 2: Mostly added. A flick view in the side bar, I can see most ('+') plus symbols rather than the others..

- Participant 3: Most of the things were added, great part of the visualization is green... beacuse the new features added as I describe before in question 1.

- Participant 4: There are mostly added "lines" (classes, methods). As I see many "plus" icons on the right side of the view. The next recurrent is modification, less deletions. At least considering the level of methods (not method lines).

- Participant 5: I think there are more methods added and more classes modified. I see on the methods that many of them appears with the symbol +. And I counted that there are more classes with no symbol, but still have changes, this means the class exist on the previous commit and was modified on the last one.

	Tool	Does it have number?	Visual variables used	Description summary
Unfamiliar CS1				
Participant 1	Thermite	Yes	None	Mostly additions, but with some changes
Participant 2	Diff	No	Icons	Mostly additions, but with some changes
Participant 3	Thermite	No	Color	Mostly additions, but with some changes
Participant 4	Diff	No	Icons	Mostly additions, but with some changes
Participant 5	Diff	No	Icons	Mostly additions, but with some changes
Unfamiliar CS2				
Participant 1	Diff	No	None	Renamed
Participant 2	Thermite	No	None	Modified and renamed
Participant 3	Diff	No	None	Movement and additions
Participant 4	Thermite	Yes	None	Added methods, modified classes
Participant 5	Thermite	Yes	None	Added methods, modified classes
Unfamiliar CS3				
Participant 1	Thermite	Yes	None	Refactoring on class hierarchy
Participant 2	Diff	No	Icons	Removals and modifications
Participant 3	Thermite	No	Color	Modifications
Participant 4	Diff	No	None	Removed methods, modified classes
Participant 5	Diff	No	Icons	Removed methods, modified classes
Unfamiliar CS4				
Participant 1	Diff	No	None	Additions, splitting refactoring
Participant 2	Thermite	No	None	More added than removed
Participant 3	Diff	No	None	Added and removed, methods removed
Participant 4	Thermite	Yes	None	Mostly additions, modified and removed elements
Participant 5	Thermite	Yes	None	Mostly additions, some lines removed

Table 7.7: Unfamiliar Pharo code: description of changes.

Presence of refactoring. The answers of the question about refactoring (*Is there a refactoring (class rename, code movement) in the commits? Please, justify.*) are summarized in Table 7.8. These answers do not display important differences according to the tool used for doing the analysis. Of particular importance, note that for change set number 4, all of the participants gave an answer on the same line: a refactoring in a class (RTMultipleData) which is split into an abstract class (RTAbstractMultiple). As an example of this summarizing procedure, here are the textual answers given by the participants for CS1:

- Participant 1: Yes. RTAboutRoassal and RTHowToContributeRoassal are moved to become subclasses of new class RTAbstractWorldMenu.
- Participant 2: Yes, we have mostly classes that are renamed. (Ex. HowToContributeRoassal -> HowToContribute).
- Participant 3: I do not thinks so, they deleted one class ... but I think it is not a refactoring.
- Participant 4: I do not see any class name changes or any kind of renaming (class or method renames).
- Participant 5: I think there is a refactor, because some of the methods of the class RTOnlineDocumentationRoassal pass to the recently added class RTOnlineDocumentation.

	Tool	Which type of refactoring (renaming, movement, tests)?
Unfamiliar CS1		
Participant 1	Thermite	Class hierarchy change
Participant 2	Diff	Renaming
Participant 3	Thermite	None
Participant 4	Diff	None
Participant 5	Diff	Renaming
Unfamiliar CS2		
Participant 1	Diff	Movement, deprecation
Participant 2	Thermite	Movement, renaming, class hierarchy changes
Participant 3	Diff	Movement
Participant 4	Thermite	Method movement
Participant 5	Thermite	Method movement into new classes
Unfamiliar CS3		
Participant 1	Thermite	Test hierarchy change
Participant 2	Diff	Test hierarchy change
Participant 3	Thermite	Deprecation
Participant 4	Diff	Test refactoring
Participant 5	Diff	None
Unfamiliar CS4		
Participant 1	Diff	Splitting RTMultipleData
Participant 2	Thermite	In RTMultipleData
Participant 3	Diff	Splitting RTMultipleData
Participant 4	Thermite	Renaming: RTMultipleData into RTAbstractMultiple
Participant 5	Thermite	Movement from RTMultipleData into RTAbstracMultipleData

Table 7.8: Unfamiliar Pharo code: presence of refactoring.

Class hierarchy description. We found that the question: *Can you describe the class hierarchy involved in the pull request/commits? Please, justify.* It was a complicated question to grade or analyze. For this reason, we are only summarizing the answers to this question in Table 7.9, and we removed this question from the Python experiment. Some of these answers seem to display a bias in favor of Git Thermite with this question. As an example, here are the textual answers given by the participants for CS1:

- Participant 1: There is a new hierarchy called RTAbstractWorldMenu. Also there are changes in RTAbstractRegularTreeLayout.

- Participant 2: No. There is nothing in the tool that tells me about the parent class or the subclasses of a class.

- Participant 3: Yes, there is one hierarchy the "Regular Tree Layout". It has only small changes.

- Participant 4: The diff does not display superclasses (class definitions) for classes without changes, so it is dificult to think of their class hierarchies. I found one as there is a class definition modification: RTAbstractElasticBoxInteraction subclass: RTBrushInteraction. There are about 8 classes where I can read superclass-subclass relation. This is only information I have about class hierarchy.

- Participant 5: It's very difcult, but I can see on the diff text only when a class is added or modified, the superclass.

	Tool	Can you describe the class hierarchy involved in the pull request/commits?
Unfamiliar CS1		
Participant 1	Thermite	Two hierachies: RTAbstractWorldMenu, and changes in RTAbstractRegularTreeLayout
Participant 2	Diff	Cannot describe.
Participant 3	Thermite	Hierarchy "Regular Tree Layout" with small changes.
Participant 4	Diff	None
Participant 5	Diff	Cannot describe.
Unfamiliar CS2		
Participant 1	Diff	Members of layout hierarchy, and members of Examples hierarchies.
Participant 2	Thermite	RTForceDrivenLayout and its subclasses.
Participant 3	Diff	RTAbstractLabeledClass and some metaclasess
Participant 4	Thermite	Three big hierarchies.
Participant 5	Thermite	I can describe it because classes are connected by edges.
Unfamiliar CS3		
Participant 1	Thermite	RTTest hierarchy is very flat. (Drawing in the answer)
Participant 2	Diff	Just the hierachy of tests because it has changes.
Participant 3	Thermite	There is one RTLayout hierarchy.
Participant 4	Diff	All tests are subclasses of RTTest
Participant 5	Diff	I cannot describe well. But tests are subclasses of RTTest.
Unfamiliar CS4		
Participant 1	Diff	The participant made a drawing.
Participant 2	Thermite	There is a class hierarchy in AbstractData class.
Participant 3	Diff	Yes, the new hierachy of abstract multiple data.
Participant 4	Thermite	RTAbstractData with multiple new classes. RTTest, RTInteraction.
Participant 5	Thermite	RTAbstractData and his subclasses.

Table 7.9: Unfamiliar Pharo code: class hierarchy description.

Difficulty of understanding changes. The results to the question: *Do you find easy to understand the changes in the commits? (strongly agree, agree, neutral, disagree, strongly disagree)* by grading with a scale between 1 (strongly disagree) and 5 (strongly agree), in-

clusive, are given in Table 7.10. In addition to the difficulty grading, the times taken by the participants for analyzing the change sets are also present in Table 7.10.

	Tool	Difficulty	Time Start	Time End	Time Delta
Unfamiliar CS1					
Participant 1	Thermite	3	17:38:00	17:50:00	00:12:00
Participant 2	Diff	2	11:31:00	11:40:00	00:09:00
Participant 3	Thermite	5	11:09:00	11:19:00	00:10:00
Participant 4	Diff	3	15:58:00	16:13:00	00:15:00
Participant 5	Diff	3	16:51:00	17:10:00	00:19:00
Unfamiliar CS2					
Participant 1	Diff	2	17:51:00	18:01:00	00:10:00
Participant 2	Thermite	5	11:40:00	11:48:00	00:08:00
Participant 3	Diff	2	11:20:00	11:32:00	00:12:00
Participant 4	Thermite	4	16:15:00	16:33:00	00:18:00
Participant 5	Thermite	4	16:10:00	16:31:00	00:21:00
Unfamiliar CS3					
Participant 1	Thermite	4	18:02:00	18:12:00	00:10:00
Participant 2	Diff	4	11:05:00	11:19:00	00:14:00
Participant 3	Thermite	5	10:43:00	10:59:00	00:16:00
Participant 4	Diff	2	16:35:00	16:47:00	00:12:00
Participant 5	Diff	2	17:12:00	17:33:00	00:21:00
Unfamiliar CS4					
Participant 1	Diff	4	18:13:00	18:20:00	00:07:00
Participant 2	Thermite	5	11:19:00	11:31:00	00:12:00
Participant 3	Diff	4	11:00:00	11:09:00	00:09:00
Participant 4	Thermite	4	16:49:00	17:02:00	00:13:00
Participant 5	Thermite	4	16:32:00	16:50:00	00:18:00

Table 7.10: Unfamiliar Pharo code: difficulty of understanding changes (1 difficult, 5 easiest), and time taken by the participants to analyze a change set.

7.4.2 Familiar code analysis

This section contains the results for the *Familiar code analysis* task of the Pharo controlled experiment. Unlike the previous task, in this task the change sets selected are different for each participant because they are analyzing familiar code (*i.e.,* source code that the participants know). Since in this task the analyzed change sets are different for each participants, they cannot be compared directly between them. However, the performance in-within a participant can still be compared, and for this reason the tables in this section are grouping the change sets by participant to facilitate this direct comparison.

Size and complexity characterization. For the first question: *Characterize the size and complexity of the commits? (trivial vs complicated) Please, justify.* The results are analyzed

and graded in Table 7.11.

	Tool	Is it complicated?	Presence of LOC?	Presence of class count	Presence of method count	Presence of examples and/or tests	Presence of example of changes	Score
Participant 1								
Familiar CS1	Thermite	0	0	0	0	0	0	0
Familiar CS2	Diff	0	0	0	0	0	0	0
Familiar CS3	Thermite	0	0	0	0	0	0	0
Familiar CS4	Diff	0	0	0	0	0	1	1
Participant 2								
Familiar CS1	Diff	1	0	0	0	0	1	2
Familiar CS2	Thermite	1	0	0	0	0	0	1
Familiar CS3	Diff	1	0	1	0	0	0	2
Familiar CS4	Thermite	1	0	0	0	0	0	1
Participant 3								
Familiar CS1	Thermite	0	0	0	0	0	0	0
Familiar CS2	Diff	1	0	0	0	0	0	1
Familiar CS3	Thermite	0	0	0	0	0	0	0
Familiar CS4	Diff	0	0	0	0	0	1	1
Participant 4								
Familiar CS1	Diff	1	0	0	0	0	0	1
Familiar CS2	Thermite	1	1	0	0	0	0	2
Familiar CS3	Diff	0	0	0	0	0	0	0
Familiar CS4	Thermite	1	1	0	0	0	0	2
Participant 5								
Familiar CS1	Thermite	0	1	1	0	0	0	2
Familiar CS2	Diff	0	0	0	0	0	0	0
Familiar CS3	Thermite	1	1	0	1	0	0	3
Familiar CS4	Diff	1	0	1	0	0	0	2

Table 7.11: Familiar Pharo code: grading of question *Characterize the size and complexity of the commits (trivial vs complicated)*

Presence of unit tests. For the second question: *Do the commits actually include unit tests? Please, justify* All of the responses given by the participant were correct, independent

of the tool used. These are the same results as the ones obtained when analyzing unfamiliar code.

Description of changes. For the third question: *How are the changes? Are things being mostly added, removed, modified, or something else? Please, justify.* The results are graded in Table 7.12.

	Tool	Does it have number?	Visual variables used	Description summary
Participant 1				
Familiar CS1	Thermite	No	None	Additions
Familiar CS2	Diff	No	None	Modifications
Familiar CS3	Thermite	No	None	Class added, method movement
Familiar CS4	Diff	No	None	Removals, additions, and modifications
Participant 2				
Familiar CS1	Diff	No	Icons	Modifications
Familiar CS2	Thermite	No	None	Additions, removals, modifications
Familiar CS3	Diff	No	Icons	Modifications, renaming
Familiar CS4	Thermite	No	None	Modifications
Participant 3				
Familiar CS1	Thermite	No	None	Additions, some modifications
Familiar CS2	Diff	No	None	Method movement
Familiar CS3	Thermite	No	None	Additions
Familiar CS4	Diff	No	None	Additions, some removed methods
Participant 4				
Familiar CS1	Diff	Yes	None	Additions
Familiar CS2	Thermite	Yes	None	Mostly additions, some class renaming
Familiar CS3	Diff	No	None	Additions, renaming (Guess)
Familiar CS4	Thermite	Yes	None	Added lines, modified classes and methods
Participant 5				
Familiar CS1	Thermite	Yes	None	Modifications
Familiar CS2	Diff	No	Icons	Modifications
Familiar CS3	Thermite	Yes	None	Additions, modified classes
Familiar CS4	Diff	No	None	Modified classes, added methods

Table 7.12: Familiar Pharo code: description of changes.

Presence of refactoring. For the fourth question: *Is there a refactoring (class rename, code movement) in the commits? Please, justify* The answers are summarized in Table 7.13.

	Tool	Which type of refactoring (renaming, movement, tests)?
Participant 1		
Familiar CS1	Thermite	None
Familiar CS2	Diff	Test refactoring
Familiar CS3	Thermite	Class splitting
Familiar CS4	Diff	None
Participant 2		
Familiar CS1	Diff	None
Familiar CS2	Thermite	Interface refactoring
Familiar CS3	Diff	Method refactoring
Familiar CS4	Thermite	API change
Participant 3		
Familiar CS1	Thermite	None
Familiar CS2	Diff	Class splitting
Familiar CS3	Thermite	None
Familiar CS4	Diff	None
Participant 4		
Familiar CS1	Diff	Class renaming
Familiar CS2	Thermite	Package renaming
Familiar CS3	Diff	Renaming (Guess)
Familiar CS4	Thermite	Renaming
Participant 5		
Familiar CS1	Thermite	None
Familiar CS2	Diff	Method renaming
Familiar CS3	Thermite	Movement
Familiar CS4	Diff	Movement

Table 7.13: Familiar Pharo code: presence of refactoring.

Class hierarchy description. The analysis of the fifth question *Can you describe the class hierarchy involved in the pull request/commits? Please, justify., about the class hierarchies*, like in the case of unfamiliar code is also omitted in this section for the same reasons. But the answers to this question are summarized in Table 7.14.

	Tool	Can you describe the class hierarchy involved in the pull request/commits?
Participant 1		
Familiar CS1	Thermite	Not any important class hierarchy involved.
Familiar CS2	Diff	No, not important for this case.
Familiar CS3	Thermite	Yes!. The S2AbstractInstrumentator hierarchy.
Familiar CS4	Diff	No, even though A is superclass of B, the method involved is not used in the hierarchy.
Participant 2		
Familiar CS1	Diff	No changes in classes to see a hierarchy.
Familiar CS2	Thermite	Yes, and there is a hierarchy involved.
Familiar CS3	Diff	I can because I know the hierarchy, and there are none here.
Familiar CS4	Thermite	Yes, and there is not hierarchy.
Participant 3		
Familiar CS1	Thermite	There is no class hierarchy.
Familiar CS2	Diff	Yes, there is a hierarchy in AbstractPaperExperiment.
Familiar CS3	Thermite	There is no hierarchy involved directly.
Familiar CS4	Diff	Not directly, however A extends from B... however since B was not modified, it does not appear on the list.
Participant 4		
Familiar CS1	Diff	I know the hierarchy because I maintain the code.
Familiar CS2	Thermite	Lot of classes in a hierarchy.
Familiar CS3	Diff	That part of the code is a mess. But I remember that DSSend<> shares the same...
Familiar CS4	Thermite	There is a new hierarchy. And there are 6 hierarchies involved in the changes.
Participant 5		
Familiar CS1	Thermite	Yes, I can see that SMGitMultiPile is a subclass of SMMultiPile.
Familiar CS2	Diff	I cannot describe it because I cannot see the superclass.
Familiar CS3	Thermite	Yes, I can describe it thanks to the edges between classes.
Familiar CS4	Diff	No, I cannot because I do not have the information.

Table 7.14: Familiar Pharo code: class hierarchy description.

Difficulty of understanding changes. As for the difficulty of understanding the changes, and the time taken for analyzing the change sets, these results are in Table 7.15.

	Tool	Difficulty	Time Start	Time End	Time Delta
Participant 1					
Familiar CS1	Thermite	5	17:12:00	17:22:00	00:10:00
Familiar CS2	Diff	5	17:22:00	17:32:00	00:10:00
Familiar CS3	Thermite	5	17:32:00	17:40:00	00:08:00
Familiar CS4	Diff	5	17:40:00	17:54:00	00:14:00
Participant 2					
Familiar CS1	Diff	2	11:49:00	11:57:00	00:08:00
Familiar CS2	Thermite	2	11:57:00	12:07:00	00:10:00
Familiar CS3	Diff	4	12:08:00	12:14:00	00:06:00
Familiar CS4	Thermite	5	12:14:00	12:22:00	00:08:00
Participant 3					
Familiar CS1	Thermite	5	11:33:00	11:36:00	00:03:00
Familiar CS2	Diff	3	11:37:00	11:42:00	00:05:00
Familiar CS3	Thermite	5	11:46:00	11:50:00	00:04:00
Familiar CS4	Diff	4	11:51:00	11:56:00	00:05:00
Participant 4					
Familiar CS1	Diff	4	17:09:00	17:18:00	00:09:00
Familiar CS2	Thermite	2	17:19:00	17:34:00	00:15:00
Familiar CS3	Diff	2	17:34:00	17:42:00	00:08:00
Familiar CS4	Thermite	4	17:43:00	17:56:00	00:13:00
Participant 5					
Familiar CS1	Thermite	5	17:36:00	17:50:00	00:14:00
Familiar CS2	Diff	3	17:51:00	18:02:00	00:11:00
Familiar CS3	Thermite	5	18:03:00	18:12:00	00:09:00
Familiar CS4	Diff	2	18:13:00	18:24:00	00:11:00

Table 7.15: Familiar Pharo code: difficulty of understanding changes, and time of analysis.

7.4.3 General impressions

Agreement questions. In the general impression task of the experiment, the first six questions are asking the participant about what is his or her level of agreement with a particular affirmation about the Git Thermite visual tool. These questions are graded in a scale between 1 (strongly disagree) and 5 (strongly agree), inclusive. The graded results of these questions are in Table 7.16. For the exact answers given by the participants in these questions, along with the justification given by the participants, refer to Appendix C.

	I find it easy to understand in general	I find easy to understand the visual variable mapping	I find easy to understand the interactions	I find it easy to navigate between visualizations	I find it useful for solving requested tasks	I find it useful for my daily job
Participant 1	5	5	5	5	4	5
Participant 2	5	5	5	5	4	3
Participant 3	4	4	4	4	4	4
Participant 4	4	4	4	3	4	2
Participant 5	5	5	5	4	5	4

Table 7.16: Pharo experiment general impression agreement questions. These questions are in a scale between 1 (strongly disagree) and 5 (strongly agree) inclusive.

Scenarios for using Git Thermite. For the question *in what scenarios and for solving which maintenance tasks would developers use Git Thermite?*, most of the answers given by the participants are about using Git Thermite for non-trivial change sets. They are also considering Git Thermite in large teams for analyzing changes proposed in a pull request. One participant mentions that he would use Git Thermite for identifying source code refactoring. For more details, see Appendix C.

Alternate tools. In the question: *What tools would you use instead of Git Thermite to retrieve the same information?* The answers given by the participants were in two lines: writing a custom tool for retrieving the same data (one participant); obtaining the same info by manually reading the textual diff (the rest of the participants). For extra details, see Appendix C.

7.4.4 NASA TLX form

The responses of the NASA TLX form for measuring cognitive load when using Git Thermite to analyze commits are in Table 7.17. The responses of the same form, but when using *Monticello textual diff* tool to analyze the commits are in Table 7.18. The differences between these two responses are presented in Table 7.19. In these responses, there is clear tendency in favor of Git Thermite for all of the task load aspects, except for the physical demand. Where no difference in terms of physical demand is a completely expected outcome for this kind of experiment. In order to test for statistical significance, we also computed the p values for the pairwise two tailed t-Student test in Table 7.19. With a 95% of confidence (*i.e.,* p value less than 0.05), we can only state that there is less mental demand on using Git Thermite, than using the *textual diff tool* for the tasks solved during this experiment.

	Mental Demand	Physical Demand	Temporal Demand	Performance	Effort	Frustration
Participant 1	13	4	9	7	12	7
Participant 2	5	1	1	4	6	6
Participant 3	15	9	11	4	14	11
Participant 4	11	11	1	1	7	1
Participant 5	10	11	10	6	9	3
Average	10.8	7.2	6.4	4.4	9.6	5.6
Standard Dev	3.77	4.49	4.98	2.30	3.36	3.85

Table 7.17: NASA TLX responses with Git Thermite in the Pharo experiment. Scale from 1 to 21, inclusive. Middle point at 11.

	Mental Demand	Physical Demand	Temporal Demand	Performance	Effort	Frustration
Participant 1	16	4	11	9	15	9
Participant 2	6	1	1	8	11	14
Participant 3	17	9	11	4	15	12
Participant 4	11	11	1	1	7	1
Participant 5	13	13	11	10	14	9
Average	12.6	7.6	7	6.4	12.4	9
Standard Dev	4.39	4.98	5.48	3.78	3.44	4.95

Table 7.18: NASA TLX responses with the *Monticello textual* diff tool in the Pharo experiment. Scale from 1 to 21, inclusive. Middle point at 11.

	Mental Demand	Physical Demand	Temporal Demand	Performance	Effort	Frustration
Participant 1	-3	0	-2	-2	-3	-2
Participant 2	-1	0	0	-4	-5	-8
Participant 3	-2	0	0	0	-1	-1
Participant 4	0	0	0	0	0	0
Participant 5	-3	-2	-1	-4	-5	-6
Average	-1.8	-0.4	-0.6	-2	-2.8	-3.4
Standard Dev	1.30	0.89	0.89	2.00	2.28	3.44
p value	0.037	0.374	0.208	0.089	0.052	0.091

Table 7.19: NASA TLX Diff responses minus Git Thermite responses. p-value is computed by using a pairwise two-tailed t-Student test.

Chapter 8

Informal Evaluation

The results of the controlled experiment presented in Chapter 7 are only partially favorable to Git Thermite in terms of having a positive user experience, and a reduced mental load. Since the experiments do not completely focus on measuring differences in terms of user experience perception, we conduct an additional exploratory informal evaluation to support and complement this positive user experience hypothesis. The objective of this informal evaluation is to assess this positive user experience by asking members of an open source community to use Git Thermite for a couple of days, and then ask them to give their feedback on using Git Thermite.

8.1 Asking an open source community

Git Thermite public release. For conducting this informal user experience evaluation, we did a public release of Git Thermite by sending an announcement message to the Pharo open source community mailing list. This announcement indicates the public GitHub page to download and install Git Thermite in a Pharo image[1], along with a demonstration video[2], and documentation on how to use Git Thermite to analyze source code change sets written in Pharo .

User feedback form. During the Git Thermite public release announcement in the Pharo community mailing list, we asked people to use Git Thermite for a period of 1.5 weeks, and to submit informal feedback through a public anonymous web form. We also told the community that if they want to, they can submit their feedback through the form more than once. The questions asked in this feedback form are the followings:

- Did you find Git Thermite useful on your work? Yes or no.
- In which tasks you find it useful?

[1]Git Thermite GitHub page: `https://github.com/ronsaldo/pharo-git-thermite`
[2]Git Thermite demonstration video: `https://youtu.be/TdBHpdyFgZE`

- Did you find Git Thermite displays accurate information for analyzing commits, yes or no?
- Did you find Git Thermite allows you to perform faster commit analysis than by using your previous tool? Yes or no.
- Did you find Git Thermite to be an improvement on the task of reviewing code? Yes or no.
- In which scenarios did you find yourself using Git Thermite more often than using textual diff?
- Any suggestions on how to improve the tool? Any critique?

8.2 Feedback from an open source community

We did not get many responses to the feedback form from the community. However, these few responses ($N = 6$), still display a favorable tendency towards Git Thermite. The remainder of this section summarizes these responses.

Git Thermite usefulness. Five participants answered that they found Git Thermite useful on their work. Only one participant answered that Git Thermite is not useful in his or her work. The participants found Git Thermite useful on the following tasks:

- "Assessing changes for releasing software."
- "Understanding the impact of commits."
- "Finding diffs in my code when committing."
- "Seeing which packages are affected."
- "Auto reviewing my code before committing, and finding some programming mistakes."

Git Thermite accuracy. All of the participants answered that Git Thermite does display accurate information for analyzing commits.

Change set analysis speed. Three of the participants found Git Thermite to be faster at analyzing commits versus the previous tool. The other three participants did not find that using Git Thermite is faster than their previous tool for analyzing commits. These responses are completely consistent with the results obtained during the controlled experiments.

Git Thermite as an improvement. All of the participants answered yes, to the question *did you find Git Thermite to be an improvement on the task of reviewing code?* These responses are directly supporting the positive user experience hypothesis of using Git Thermite for change set analysis.

Scenarios for using Git Thermite. In the question about which scenarios a developer would use Git Thermite for analyzing source code change sets, we got the following responses:

- "Getting an overview of medium sized commits."
- "To check the number of affected elements by the change set."
- "To see if changes are affecting several unrelated packages."
- "Auto reviewing my code before committing."
- "Looking at changes from other people on foreign code."

Suggestions and critiques. The main critique that we received is that when analyzing large change sets with more than 400 classes, the visualizations are so large that they do not fit in the screen. The suggestion received along with this critique says that additional methods of filtering would be something that could help in handling this large change set. Another critique is in how Git Thermite is displaying only comments of the last commit when analyzing a range of commits. Some users are criticizing long visualization building time, along with the lack of a progress bar indicating when the visualization is going to be ready.

Chapter 9

Discussion

This chapters performs a further analysis and discussion of the experimental results that are presented in Chapter 7 and Chapter 8. Section 9.1 of this chapter confronts the research hypotheses that are constructed in Section 6.2 with the experimental data. Each one of these sections discusses one of the hypothesis by further analyzing the experimental data. Section 9.2 discusses the threats to validity of our experimental results, and Section 9.3 discusses the limitations of our implementation. The final conclusions from this gathered data and this discussion are presented in the next chapter.

9.1 Hypothesis testing

Chapter 6 introduces a number of null hypotheses with their corresponding alternate hypotheses that are described in Table 6.1. By contrasting these pairs of hypotheses with the experimental data, we decide on whether to keep a null hypothesis, or whether the data supports an alternate hypothesis.

Due to the small sample size in our experiment (five Pharo practitioners), we are not able to perform proper statistic significance tests. Instead, for this discussion we perform a qualitative analysis of the results.

9.1.1 Differences in change set analysis time

In terms of change set analysis time, we formulated the following null and alternate hypotheses:

- Null hypothesis: There is no *time difference* between using *Git Thermite* and *textual diff*.
- Alternate hypothesis: Commit analysis is significantly *faster* with *Git Thermite* than with *textual diff*.

Git Thermite			Diff		
Participant	Change set	Time	Participant	Change set	Time
P1	CS1	12	P2	CS1	9
P3	CS1	10	P4	CS1	14
P2	CS2	8	P5	CS1	19
P4	CS2	18	P1	CS2	9
P5	CS2	21	P3	CS2	12
P1	CS3	10	P2	CS3	14
P3	CS3	16	P4	CS3	12
P2	CS4	12	P5	CS3	21
P4	CS4	13	P1	CS4	6
P5	CS4	17	P3	CS4	8
Average		13.80	Average		12.80
Standard Deviation		4.24	Standard Deviation		4.52

Table 9.1: Unfamiliar Pharo code: experimentation time analysis. Times are in minutes.

The average time taken during the Pharo experiment tasks of analyzing change sets with Git Thermite is 13.80 minutes, and the standard deviation is 4.24 minutes. In contrast, the average time when using the *textual diff* tool for analyzing is 12.80, and the standard deviation is 4.52 (See Table 9.1). These results seem to display an average advantage of one minute in favor of using the *diff textual tool* instead of Git Thermite for analyzing changes. However, since the difference in the means is completely contained in the standard deviation by a margin of four times, it is quite clear that this difference in average time is not statistically significant. This small difference can be attributed to random variations in the experiment, and the small size of the sample set. This variation could also be attributed to the fact that the participants are more familiar with the textual diff tool, but this familiarity is not expected to provide a substantial advantage in favor of the textual diff tool because of the large size of the change sets.

Git Thermite			Diff		
Participant	Change set	Time	Participant	Change set	Time
P1	CS1	10	P1	CS2	10
P1	CS3	8	P1	CS4	14
P2	CS2	10	P2	CS1	8
P2	CS4	8	P2	CS3	6
P3	CS1	3	P3	CS2	5
P3	CS3	4	P3	CS4	5
P4	CS2	15	P4	CS1	9
P4	CS4	13	P4	CS3	8
P5	CS1	14	P5	CS2	11
P5	CS3	9	P5	CS4	11
Average		9.40	Average		8.70
Standard Deviation		3.95	Standard Deviation		2.91

Table 9.2: Familiar Pharo code: experimentation time analysis. Times are in minutes.

In the case of familiar code analysis tasks during the Pharo experiment, the average analysis time when using Git Thermite is 9.40 minutes, and the standard deviation is 3.95 minutes. When using the *textual diff tool*, the average time of analysis is 8.70 minutes, and the standard deviation is 2.91 minutes (See Table 9.2). These results are consistent with the case of analyzing unfamiliar code, in terms of 0.70 minutes (or 42 seconds) of average in favor to using the *textual diff tool*. Like in the previous case, this average difference is quite small in comparison with the standard deviation, so this difference is not statistically significant.

In other words, there is no significant difference in using Git Thermite, or the *textual diff tool* for analyzing change sets in terms of time. For this reason, we keep accepting the null hypothesis. Since the usage of Git Thermite is also not faster than using the *textual diff tool*, we also reject the alternate hypothesis.

9.1.2 Change set analysis precision

In terms of change set analysis precision, or how similar are the answers given by the participants according to the tool used for doing the analysis, we formulated the following null and alternate hypotheses:

- Null hypothesis: There is no difference in terms of *analysis precision* between using *Git Thermite* and *textual diff.*
- Alternate hypothesis: The source code analyses that are made with *Git Thermite* have significantly more *precision* than the analyses that are made with *textual diff.*

In the unfamiliar code analysis task of the Pharo controlled experiment there are no marked tendencies that favor one tool, or another in the questions that are about describing change sets. These change set characterization questions are the followings:

- *Characterize the size and complexity of the commits? (trivial vs complicated) Please, justify.* See Table 7.6 for a grading of this question in terms of the presence of specific elements in the answers given by the participants.
- *Do the commits actually include unit tests? Please, justify* All the participants answered this question correctly, independently of the tool.
- *How are the changes in the commits? Are things being mostly added, removed, modified, or something else? Please, justify.* This question is graded in Table 7.7, which does not show a tendency in favor of one of the tools.
- *Is there a refactoring (class rename, code movement) in the commits? Please, justify* The answers to this question, summarized in Table 7.8, do not display important differences according to the tool used for doing the analysis. Most of the answers are duplicated, but with a different tool used for doing the analysis.

Since these questions are about describing and characterizing a change set in different aspects; the answers given to these questions can be used to measure the precision of a tool for analyzing a change set. We cannot state that one tool has more precision than the other one for analyzing change sets because of the lack of important differences in the answers

given by the participants to these questions. Therefore, we keep the null hypothesis about analysis precision, and reject our alternate hypothesis.

9.1.3 Intention of changes inference

In terms of *inferring the intention of changes*, we formulated the following null and alternate hypotheses:

- Null hypothesis: The intention of changes inferred with *Git Thermite* is the same intention inferred with *textual diff.*
- Alternate hypothesis: The intention of changes inferred with *Git Thermite* is more detailed than the intention inferred with *textual diff.*

The questions related to this hypothesis administered during the unfamiliar code analysis task of the Pharo controlled experiment are the followings:

- *Characterize the size and complexity of the commits? (trivial vs complicated) Please, justify.*
- *How are the changes in the commits? Are things being mostly added, removed, modified, or something else? Please, justify.*
- *Is there a refactoring (class rename, code movement) in the commits? Please, justify.*

These questions are the same, analyzed for testing the hypotheses about *change set analysis precision* in Section 9.1.2. Since the answers to these questions do not display differences according to the tool used for the change set analysis; we cannot state a difference between Git Thermite, and the *textual diff* for *inferring the intention of changes* for a given source code change set. We keep the null hypothesis and reject the alternate hypothesis because of this.

9.1.4 Cognitive load

In terms of cognitive load, we formulated the following null and alternate hypotheses:

- Null hypothesis: The cognitive load of doing an analysis with *Git Thermite* is the same as doing the analysis with *textual diff.*
- Alternate hypothesis: There is less measurable cognitive load by using *Git Thermite* instead of using *textual diff.*

NASA TLX form answers. In the case of the Pharo experiment, the difference between the answers given by the participants to the NASA TLX form for Git Thermite, and the form for the textual diff tool, mark a clear tendency in favor of Git Thermite (See Table 7.19) for all of the points. In the case of the Python experiment attempt, this same tendency in favor of Git Thermite seems to repeat (See Table A.18). However we cannot state statistical

significance in the case of the Python experiment because of the low number of samples. The p value for the pairwise two tailed t-Student test reflect a statistical significance for the difference in terms of *mental demand* in the Pharo experiment ($N = 5$) for a 95% confidence interval. If we accept a 90% confidence interval, then the differences in terms of *performance*, *effort* and *frustration* can also be deemed with statistical significance. Despite these statistically significant p values, due to the very small $N = 5$ value, these results are difficult to generalize.

Participants general impressions. The answers to the general impressions questions given by the participant tend to favor Git Thermite. In particular, some answers to the question *I find it easy to understand in general (strongly agree, agree, neutral, disagree, strongly disagree)* say that the tool definitely helped them understand the changes; that the colors and the elements present in the visualization help explain the changes in the code. These responses by the participants support the hypothesis that Git Thermite can be less demanding on the users than the *textual diff tool*.

We have to reject the null hypothesis that there are no differences in terms of cognitive load because of the evidence found during the previous analysis. Since all of the measured differences in the NASA TLX are in favor of Git Thermite, we accept the alternate hypothesis: There is less measurable cognitive load by using *Git Thermite* instead of using *textual diff.*

9.2 Threats to validity

Experiment questions. One threat to the validity of our experimental results is on the selection of the experimental question. The questions used in our experiment are selected with the purpose of trying to reflect real world questions asked by an open source project integrator when deciding to accept a contribution. The problem is that these questions tend to be broad in scope, so the responses are challenging to analyze and quantify. There is a strong possibility that more focused questions could yield more significant differences between Git Thermite and the *textual diff tool*. However, asking very focused questions can introduce a bias related to measuring a completely artificial laboratory setting that is completely different from a real word usage scenario. Properly balancing the questions with the objective of running a completely rigorous controlled experiment is beyond the scope of this master thesis.

Sample size. Another threat to the validity of our experimental results is related to the sample size. The small sample size in our experiments makes it hard to generalize the results to other populations. The statistical significance found in the reduced mental demand during the Pharo experiment ($N = 5$) can be attributed to the fat tails of the t-Student distribution. The results of the Python experiments are weak. The Python experiment is weak because of the small sample size ($N = 2$), and the experimentation problems due to the lack of a proper Python parser. However, the Python controlled experiment still seems to support the

hypothesis of a reduced cognitive load when using Git Thermite instead of a *textual diff tool*. The sample size threat to validity can be reduced by increasing the sample size.

Programming language bias. Differences between analyzed programming languages can also be another source of bias, and another threat to the validity of our experimental results. Different programming languages can have a different meta-model, which can be unsuitable for displaying with our visualization. Other differences related to the programming languages are in the culture on how to program and solve problems by the practitioners. In order to reduce this threat to validity, we tried to perform a controlled experiment with Python practitioners, in addition to the first experiment performed with Pharo practitioners. We failed at performing this Python experiment properly due to the lack of a proper Python parser written in Pharo. However, the partial results from the Python controlled experiment are still favorable to Git Thermite in terms of user experience perception, which is independent of the problems produced by the Python parser. To reduce the threat even more, further experimenting with completely different languages is required.

Open source vs closed source. Another source of threat to the validity of the results is in the differences between open source and closed source software. The design of the visual tool is made in the context of facilitating an open source project integrator in reviewing contributions. The workflow used by closed source projects is not necessarily the same as the one used by open source projects. Doing a proper evaluation of Git Thermite in a closed source environment can be complicated because of confidentiality reasons. Because of this, properly assessing the usefulness of a visual tool such as Git Thermite in a closed source environment, with a rigorous controlled experiment, is beyond the scope of this master thesis.

Bias of answering. There is a potential source of bias from the participants on the mere fact of being a participant of both: our controlled experiments and the informal evaluation. If the answers given by the participants are biased, they would be strongly biased in favor of Git Thermite. The answers that we received from the participants do not show this kind of bias, and in fact the answers are in some aspect neutral (*e.g.,* lack of qualitative differences in some responses), or even against the usage of Git Thermite (*e.g.,* less time taken by using the textual diff tool for analyzing commits). For this reason, we believe that we obtained honest answers from all of the participants because of the lack of this expected bias.

9.3 Limitations in our implementation

Full history navigation. One limitation in our current implementation of Git Thermite is in the lack of support for full source code history navigation. In the first versions of Git Thermite we tried to add support for history navigation with the commit tree visualization. During the successive visualization testing and refinement design loop, we only focused on the main structural source code changes visualization, and the business card as an entry point with metrics about the change set. Since we focused on evaluating these visualizations, we

neglected the commit tree visualization, and we did not implement proper support full source code history navigation. One possibility for fixing this limitation consists in following these steps: adding back the commit tree visualization; improving on the commit tree visualization; allowing to select a single commit or multiple commits in the commit tree visualization; and opening the rest of the Git Thermite visualizations but using this selection of commits.

Better code movement detection. Another limitation in Git Thermite is in the detection of code movement. Currently we are just comparing the whole source code of methods after trimming leading and trailing white spaces. This technique works fine for detecting completely moved methods, and functions. However, this technique is not capable of detecting movement of code that is partially changed in the process, or renamed methods. Unfortunately, doing proper detection of code movement is a complicated and hard problem. Solving this limitation requires implementing a code similarity heuristic, which has to be fine tuned to specific thresholds in order to prevent a high number of false positives.

Mixed programming language analysis. The main programming language used in a source code repository has to be specified to Git Thermite before analyzing a change set. This is a limitation in our implementation of Git Thermite in terms of analyzing a change set that is written in multiple programming languages. Web applications are typical examples of projects written in multiple programming languages, so this limitation affects several real world projects. Solving this limitation requires implementing support for inferring the programming languages used by a project, and adding support in Git Thermite for more programming languages.

Users of affected code. A common comment received by the participants of the controlled experiments, including the pilots, is that they would like to know the affected parts of the system in terms of the users of the changed code. Our visualization only represents the modified code, and does not represent whether a changed method is only used in one place, or all over around the system. Solving this limitation requires doing a dependency analysis over the whole repository, and designing a proper way to display the affected dependents.

Chapter 10

Conclusions

Research objective. In terms of the research objectives that are introduced in Section 1.4, we accomplished all of them:

- *Define or use existing visual metaphors to precisely convey information about changes.* This objective is accomplished with the design and implementation of the Git Thermite visualization.
- *Integrate metrics and visualizations about changes in a single tool, for enabling a workflow for assessing the different aspects of a software change.* This objective is also accomplished with Git Thermite.
- *Validate the tool with people for testing the hypothesis.* We accomplished this objective with the design and execution of the Git Thermite controlled experiment.

Research hypothesis testing. In Section 1.3 we introduced the following research hypothesis:

By using a visual analysis and reporting tool that displays metrics, the changes to the structure of the software and the source code, in addition to the information presented by the textual diff, it is possible to facilitate code reviewing process in terms of time, comprehension of the changes and cognitive load to the reviewers.

We divided this large hypothesis in four pairs of null and alternate hypotheses in Section 6.2. We tested these hypotheses in Section 9.1 by contrasting with the experimental data gathered during the execution of the Pharo controlled experiment exposed in Chapter 7. The results for each one of these hypotheses are the followings:

- There are no differences in terms of change set analysis time. (See Section 9.1.1).
- There are no differences in terms of change set analysis precision. (See Section 9.1.2).
- There are no differences in terms of inferring the intentions for a given change set. (See Section 9.1.3).
- There are some differences in terms of cognitive load. (See Section 9.1.4).

These results only partially support the full research hypothesis: a reduced cognitive load when using Git Thermite instead of the textual diff tool when analyzing a source code change set. The other aspects of our research hypotheses are being disproved by our experimental results. In other words, we did not find differences between using a visual tool (Git Thermite) or the textual diff tool for analyzing source code change sets in terms of time, and the comprehension of changes.

Improved user experience with a visual tool. With our experiment, we learned that the usage of a visual tool in opposition to the traditional textual only tool, does not not necessarily provide measurable advantages in terms of performing a task faster, or with better results. However, we still are finding an advantage in terms of user experience. This subjective advantage can be complicated to measure, and since our controlled experiments are not completely focused on measuring these user perception difference, we are only capturing a glimpse through the usage of the NASA TLX form, and the comments given by the participants. This small improvement is still important for reducing the mental stress of software developers, especially when they have to perform a tedious task such as assessing a large open source contribution.

Git Thermite as a complement. Since we did not find many advantages to using Git Thermite, we still suggest it as a complement to the textual diff tool. We suggest using Git Thermite specifically for analyzing large and non-trivial change sets. For trivial change sets of one or two lines of code, Git Thermite will not provide any useful information (*e.g.,* a visualization with only a single element), so using the textual diff tool should be faster.

Future work

Solving some limitations. One future line of work is on solving some of the limitations in our current implementation of Git Thermite. In particular, we would like to solve the limitation on the lack of support for projects with mixed languages, in order to use Git Thermite for analyzing common web applications. Another limitation that we would like to keep working on is that of adding support for displaying the users of the changed code. We believe that with that feature in particular, we could identify differences when using Git Thermite and the textual diff tool.

Integration on GitHub. GitHub is one of the most popular collaborative open source platforms. Currently we have support for analyzing a GitHub pull request with Git Thermite, but we could add support the other way around: to use Git Thermite inside of GitHub. GitHub provides web hooks for integrating third party application (*e.g.,* bots) that receive notifications when a commit is performed, a pull request is made, or a comment in a pull request or commit is made by a user. These application can also react by posting comments to the pull requests, or commits in GitHub. By using these GitHub facilities we could provide Git Thermite as a GitHub application, and use it with real world open source projects.

Further experimentation. The differences on cognitive load were not the focus in the design of our controlled experiment. Since this is the measured aspect where we found the most differences in favor to Git Thermite, it makes sense to perform another experiment focused on testing the following hypothesis: *When analyzing source code change sets, there is less cognitive load when using Git Thermite than by using a textual diff tool.* For this new controlled experiment we would like to incorporate elements from the methodology used by Merino *et al.* for evaluating 3D software visualizations with augmented reality [31].

Solving the problems presented by our controlled experiment require complementary experimentation. Doing a larger scale controlled experiment, with a larger sample size, and involving a larger number of programming languages is a future line of work. Getting a large sample size by itself is a time consuming job. We plan on involving practitioners from other communities, such as the Java community in a future experiment. With a larger sample size, it becomes possible to use stronger statistical tests for significance on the experimentation results.

Bibliography

[1] Michele Lanza and Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, 2003.

[2] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: A controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 551–560. ACM, 2011.

[3] Teseo Schneider, Yuriy Tymchuk, Ronie Salgado, and Alexandre Bergel. Cuboidmatrix: Exploring dynamic structural connections in software components using space-time cube. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 116–125. IEEE, 2016.

[4] Verónica Uquillas Gómez, Stéphane Ducasse, and Theo D'Hondt. Visually characterizing source code changes. *Science of Computer Programming*, 98:376–393, 2015.

[5] Verónica Uquillas Gómez, Stéphane Ducasse, and Theo D'hondt. Visually supporting source code changes integration: the torch dashboard. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 55–64. IEEE, 2010.

[6] Andrew McNair, Daniel M German, and Jens Weber-Jahnke. Visualizing software architecture evolution using change-sets. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 130–139. IEEE, 2007.

[7] 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.

[8] Brian De Alwis and Jonathan Sillito. Why are software projects moving from centralized to decentralized version control systems? In *Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, pages 36–39. IEEE Computer Society, 2009.

[9] Jason Tsay, Laura Dabbish, and James Herbsleb. Influence of social and technical factors for evaluating contribution in github. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 356–366, New York, NY, USA, 2014. ACM.

[10] Daricélio Moreira Soares, Manoel Limeira de Lima Júnior, Leonardo Murta, and Alexandre Plastino. Acceptance factors of pull requests in open-source projects. In *Proceedings*

of the 30th Annual ACM Symposium on Applied Computing, SAC '15, pages 1541–1546, New York, NY, USA, 2015. ACM.

[11] Jesse James Garrett. *Elements of user experience, the: user-centered design for the web and beyond*. Pearson Education, 2010.

[12] Karel Vredenburg, Ji-Ye Mao, Paul W Smith, and Tom Carey. A survey of user-centered design practice. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 471–478. ACM, 2002.

[13] Brian M Winn. The design, play, and experience framework. In *Handbook of research on effective electronic gaming in education*, pages 1010–1024. IGI Global, 2009.

[14] Paul Eggert, Mike Haertel, David Hayes, Richard Stallman, and Len Tower. *GNU diff user manual*. Free Software Foundation, Inc.

[15] About comparing branches in pull requests. `https://help.github.com/en/articles/about-comparing-branches-in-pull-requests`. Accessed: 2019-09-12.

[16] Comparing commits across time. `https://help.github.com/en/articles/comparing-commits-across-time`. Accessed: 2019-09-12.

[17] Andrew P Black, Oscar Nierstrasz, Stéphane Ducasse, and Damien Pollet. *Pharo by example*. Lulu.com, 2010.

[18] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Deep Into Pharo*. Lulu.com, 2013.

[19] About pull requests. `https://help.github.com/en/articles/about-pull-requests`. Accessed: 2019-09-12.

[20] Colin Atkinson and Thomas Kuhne. Model-driven development: a metamodeling foundation. *IEEE software*, 20(5):36–41, 2003.

[21] Verónica Uquillas Gómez, Stéphane Ducasse, and Theo D'Hondt. Ring: a unifying meta-model and infrastructure for smalltalk source code analysis tools. *Computer Languages, Systems & Structures*, 38(1):44–60, 2012.

[22] Richard Wettel. *Software systems as cities*. PhD thesis, Università della Svizzera italiana, 2010.

[23] Benjamin Bach, Emmanuel Pietriga, and Jean-Daniel Fekete. Visualizing dynamic networks with matrix cubes. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 877–886. ACM, 2014.

[24] Fernanda B Viégas and Martin Wattenberg. Timelines tag clouds and the case for vernacular visualization. *interactions*, 15(4):49–52, 2008.

[25] Verónica Uquillas Gómez. *Supporting Integration Activities in Object-Oriented Applica-*

tions. PhD thesis, Lille University of Science and Technology, France, 2012.

[26] Andy Pryke, Sanaz Mostaghim, and Alireza Nazemi. Heatmap visualization of population based multi objective algorithms. In *International Conference on Evolutionary Multi-Criterion Optimization*, pages 361–375. Springer, 2007.

[27] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*, pages 285–311. Springer, 2008.

[28] Sandra G Hart and Lowell E Staveland. Development of NASA-TLX (task load index): Results of empirical and theoretical research. In *Advances in psychology*, volume 52, pages 139–183. Elsevier, 1988.

[29] Teseo Schneider, Yuriy Tymchuk, Ronie Salgado, and Alexandre Bergel. Cuboidmatrix: Exploring dynamic structural connections in software components using space-time cube. In *Software Visualization (VISSOFT), 2016 IEEE Working Conference on*, pages 116–125. IEEE, 2016.

[30] Alexandre Bergel. *Agile Visualization.* Lulu.com, 2016.

[31] Leonel Merino, Alexandre Bergel, and Oscar Nierstrasz. Overcoming issues of 3d software visualization through immersive augmented reality. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 54–64. IEEE, 2018.

Appendix A

Python controlled experiment attempt

After the Pharo controlled experiment, we conducted a second one with Python practitioners. Since we did not find significant differences between using Git Thermite, and the Monticello textual diff tool, we expected to find more differences with Python practitioners.

Shortening the experiment. Unlike the Pharo experiment, we decided to simplify and shorten the Python experiment, because some of the session of the Pharo controlled experiment lasted up to three hours. Long experimentation times can be a problem, especially for finding willing participants from the industry. For this reason we made the following changes: analyze two familiar change sets instead of four; analyze two unfamiliar change sets instead of four.

A.1 Questions

A.1.1 Personal questions

The questions used in the first task in the Python experiment are the followings:

1. How many years of experience do you have in programming?
2. How long have you been programming in Python?
3. Which other programming languages do you know?
4. What is your highest achieved academic degree? (optional) Choose between: Undergraduate, professional practitioner, master, PhD, and other (Specify).
5. How familiar are you with using software visualizations? Choose one of the followings:
   - Zero. I have never used a visualization.
   - Low. I almost never use a visualization.
   - Medium. I regularly use a visualization.
   - High. I frequently use a visualization, or it is part of my daily job.

6. Which software visualization?
7. How often do you use Git? On a scale from 1 (Never) to 5 (Very often).
8. How familiar are you with using Git? On a scale from 1 (Not familiar at all) to 5 (Very familiar).
9. Which version control system do you use?
10. How often do you use GitHub? In a scale from 1 (Never) to 5 (Very often).
11. How familiar are you with using GitHub? In a scale from 1 (Not familiar at all) to 5 (Very familiar).
12. Do you review source code changes? If yes, how?
13. When you need to perform a textual diff, which tool do you use for doing the diff?
14. What are the limitations you see in Git diff tool that you do use?
15. How do you integrate commits made by a developer other than you?
16. How do you verify a commit made by a developer other than you?
17. In which kinds of scenarios do you need to review the history? For example, one typical scenario for doing this is having a bug regression or a performance regression.
18. How do you navigate in the history of a project?
19. Which application do you wish to analyze?
20. How do you qualify your expertise for the application used in this study? In a scale from 1 (Not much knowledge) to 5 (Expert),

A.1.2 Source code change set questions

The following is the list of questions from the *source code change set analysis* tasks, in the Python experiment:

1. How much of the application is touched by the changes in terms of classes? Choose one of: Few (1-2), medium (3-6), and many (more than 6).
2. How much of the application is touched by the changes in terms of methods/functions? Choose one of: Few (1-5), medium (6-15), and many (more than 15).
3. What is the complexity of the touched parts

Like in the Pharo experiment, these questions are used for both familiar and unfamiliar code.

A.1.3 Questions about the user general impressions on the tool

The following is the list of questions used for the task about the user *general impressions* in the Python experiment:

1. I find it easy to understand in general: (strongly agree, agree, neutral, disagree, strongly disagree) Please, justify

2. I find it easy to understand the visual variable mapping: (strongly agree, agree, neutral, disagree, strongly disagree) Please, justify
3. I find it easy to understand the interactions: (strongly agree, agree, neutral, disagree, strongly disagree) Please, justify.
4. I find it easy to navigate between visualizations: (strongly agree, agree, neutral, disagree, strongly disagree) Please, justify.
5. I find it useful for solving requested tasks: (strongly agree, agree, neutral, disagree, strongly disagree) Please, justify.
6. I find it useful for my daily job: (strongly agree, agree, neutral, disagree, strongly disagree) Please, justify.
7. In what scenarios and for solving which maintenance tasks would developers use Git Thermite? Please, justify.
8. What tools would you use instead of Git Thermite to retrieve the same information? Please, justify.
9. Do you have any suggestions on how to improve the visual report or the tool itself? Any critique? Please, justify.

A.2 Pilot

For the Python experiment we conducted a pilot with two students. This pilot experiment was used to test the simplified questions. Overall, we did not detect problems during the execution of this pilot experiment. For this experiment, we selected two pull requests from Keras[1]. The change sets used in the Python experiments are displayed in Table A.1.

Change set number	Project	Pull request number
CS1	Keras	10130
CS2	Keras	10781

Table A.1: Selected change sets for the Python experiment.

A.3 Execution

We conducted the final Python experiment with two professional practitioners from the industry. See Table A.2 for the background of the python practitioners.

[1]Keras is a high-level API for deep learning. `https://github.com/keras-team/keras`

	Programming years of experience	Python programming experience	Highest academic degree
Participant 1	7	7 years	Professional practitioner
Participant 2	2	1.5 years	Master

Table A.2: Python experiment participants background.

Experience with visualizations, Git and GitHub. The participants of this experiment reported that they almost do not have experience with using visualizations (See Table A.3). They also reported that they have a moderate knowledge of Git and GitHub (See Table A.4).

	Familiarity using visualizations	Which visualizations
Participant 1	Low: I almost never use a visualization	Network visualizations from Github / Gitlab, Git UIs
Participant 2	Zero: I have never used a visualization	

Table A.3: Python participant experience with visualizations

	Git usage frequency	Git familiarity	Other VCS	GitHub usage frequency	GitHub familiarity
Participant 1	5	5	None	5	5
Participant 2	3	3	None	3	2

Table A.4: Python participants experience with Git and GitHub.

Textual diff tool. The participants reported that for doing a textual diff, they normally use the command line *git diff* tool, an UI from the text editor, or the tool provided by a collaborative platform such as GitHub and GitLab. See Table A.5 for more details.

	Tool used for textual diff	Textual diff tool limitations
Participant 1	Command line or UIs from code editors.	Only from the text differences is it difficult to realize the real extent of the changes. Small batches of changes help me with that.
Participant 2	git diff in terminal o GitLab.	

Table A.5: Textual diff tool used by the Python practitioners

Experience with source code reviewing. The answered questions related to source code reviewing by the participants of this experiment are in Table A.6.

	Do you review source code changes? If yes, how?	How do you integrate commits made by a developer other than you?	How do you verify a commit made by a developer other than you?
Participant 1	Git commands (diff), network/ branches visualizations	Merge, rebase or start a new branch from develop making a merge to this new branch if there are too many changes	Check the published changes on a local machine. Review the main changes, but mainly the tests
Participant 2		I just perform a git merge, and nothing else.	I do a git merge in a local machine, then I check if everything is working (tests...)

Table A.6: Python participants experience with source code reviewing.

Change set assignment. The change sets for unfamiliar code analysis tasks assigned to the Python practitioners are described in Table A.7.

	First change set	Second change set
Participant 1	CS1 Thermite	CS2 Diff
Participant 2	CS2 Thermite	CS1 Diff

Table A.7: Python experiment change set assignment

A.4 Experimentation problems

Parsing problem. During the execution of the Python controlled experiment, we noticed a bug in the implementation of Git Thermite. The tool tended to display wrong visualizations because it tended to lump together multiple methods into a single method. This problem occurs because we are not using an actual Python parser in our implementation of Git Thermite. Instead of doing actual parsing of Python, we are relying on the indentation of Python code for extracting the boundaries of class, methods a functions in a Python source file. In several of our tests, this technique seemed to work, however this technique failed during the execution of this experiment with real world Python code. This problem can be solved by using a proper Python code, however, we are lacking an existent implementation of a suitable Python parser for using in Git Thermite that is also implemented in Pharo. The usage of indentantion as a structural element in Python makes it harder to use a parser

generator with a Python grammar. Doing another Python parser to fix this parsing problem is beyond the scope of this thesis because of this last reason.

A.5 Results

A.5.1 Unfamiliar code analysis

Size and complexity characterization. For the section of characterizing the size and complexity of the changes, this questions is divided into multiple questions that are easier to evaluate in this experiment. The answers given by the Python practitioners are in Table A.8.

	Tool	How much of the application is touched by the changes in terms of classes?	How much of the application is touched by the changes in terms of methods/functions?	What is the complexity of the touched parts?
Unfamiliar CS1				
Participant 1	Thermite	Medium	Many	Medium complexity
Participant 2	Diff			Medium complexity
Unfamiliar CS2				
Participant 1	Diff	Medium	Medium	Medium complexity
Participant 2	Thermite			Medium complexity

Table A.8: Unfamiliar Python code: size and complexity.

Unit test detection. The question about the presence of unit tests has divergent results. These results can be attributed to the lack of proper unit test detection in the implementation of Git Thermite, since the wrong answer is due to the user guiding his or herself on the colors used to highlight the word *test*. For details see Table A.9.

	Tool	Do the commits actually include unit tests? Please, justify	Is the answer correct?
Unfamiliar CS1			
Participant 1	Thermite	Yes. For this I check the files and there is one file with tests (pytest)	Yes
Participant 2	Diff		
Unfamiliar CS2			
Participant 1	Diff	Yes. They are using pytest (I was using the search option in the browser after opening several pages with the realted commits).	Yes
Participant 2	Thermite	No, nothing in green	No

Table A.9: Unfamiliar Python code: Presence of tests

Description of changes. Table A.10 highlights the answers to the questions that describe the changes in the change set.

	Tool	Are the changes in the commit composed mostly of?	What are the changes in the commits? Please, justify	Is there a refactoring (class rename, code movement) in the commits? Please, justify
Unfamiliar CS1				
Participant 1	Thermite	Additions	Add methods for some transformations	Seems that this is not the case. There were no "lines" showing a possible refactor.
Participant 2	Diff			I guess there is a refactoring, due to the name of the commit.
Unfamiliar CS2				
Participant 1	Diff	Additions	Adding several methods	Yes. I can see that from the conversation.
Participant 2	Thermite	Removals		No, nothing in blue

Table A.10: Unfamiliar Python code: description of changes.

Difficulty of understanding changes. The results to the question: *Do you find it easy to understand the changes in the commits? (strongly agree, agree, neutral, disagree, strongly disagree)* By grading with a scale of 1 and 5, the results are given in Table A.11. In addition to the difficulty grading, the times taken by the participants for analyzing the change sets are also present in Table A.11.

	Tool	Difficulty	Time Start	Time End	Time Delta
Unfamiliar CS1					
Participant 1	Thermite	4	15:44:00	15:56:00	00:12:00
Participant 2	Diff		15:48:00	15:57:00	00:09:00
Unfamiliar CS2					
Participant 1	Diff	4	15:57:00	16:03:00	00:06:00
Participant 2	Thermite	3	15:44:00	15:48:00	00:04:00

Table A.11: Unfamiliar Python code: difficulty of understanding changes, and time of analysis.

A.5.2 Familiar code analysis

Size and complexity characterization. The answers to the questions that are characterizing the size and complexity are present Table A.12.

	Tool	Is this PR/-commit representative of the usual changes that are made to this software?	How much of the application is touched by the changes in terms of classes?	How much of the application is touched by the changes in terms of methods/functions?	What is the complexity of the touched parts?
Participant 1					
Familiar CS1	Thermite	Yes	Few	Few	Low complexity
Familiar CS2	Command line textual diff	No	Few	Few	Low complexity
Participant 2					
Familiar CS1	Thermite	Yes	Many	Medium	Medium complexity
Familiar CS2	GitHub PR diff	Yes	Few	Medium	Low complexity

Table A.12: Familiar Python code: size and complexity.

Description of changes. The answers to the question that are describing the changes in the change sets are in Table A.13.

	Tool	Are the changes in the commit composed mostly of?	How are the changes in the commits? Please, justify	Is there a refactoring (class rename, code movement) in the commits? Please, justify
Participant 1				
Familiar CS1	Thermite	Other changes: the number of additions and removals were similar, because of some updates in the code.	It was actually an update of several methods / parameters	Not in this case.
Familiar CS2	Command line textual diff	Other changes: kind of the same number of additions and removals.	Updates in methods because of some libraries	Not really. There are small changes in some lines in terms of parameters.
Participant 2				
Familiar CS1	Diff	Removals		No, there is no blue.
Familiar CS2	GitHub PR diff	Removals		

Table A.13: Familiar Python code: description of changes.

Difficulty of understanding changes. The answers to the question: *Do you find it easy to understand the changes in the commits? (strongly agree, agree, neutral, disagree, strongly disagree)* Along with the times taken by the participants to analyze the change sets are present in Table A.14.

	Tool	Difficulty	Time Start	Time End	Time Delta
Participant 1					
Familiar CS1	Thermite	5	15:25:00	15:33:00	00:08:00
Familiar CS2	Command line textual diff	2	15:35:00	15:39:00	00:04:00
Participant 2					
Familiar CS1	Thermite	4	15:24:00	15:34:00	00:10:00
Familiar CS2	GitHub PR diff	3	15:37:00	15:41:00	00:04:00

Table A.14: Familiar Python code: difficulty of understanding changes, and time of analysis.

A.5.3 General impressions

Agreement questions. In the general impression task of the experiment, the first six questions ask the participants about their level of agreement with a particular affirmation about the Git Thermite visual tool. These questions are graded on a scale of 1 (strongly disagree) and 5 (strongly agree), inclusive. The graded results of these questions are in Table A.15. For the exact answers refer to Appendix C.

	I find it easy to understand in general	I find it easy to understand the visual variable mapping	I find it easy to understand the interactions	I find it easy to navigate between visualizations	I find it useful for solving requested tasks	I find it useful for my daily job
Participant 1	4	4	3	4	4	3
Participant 2	4	5	4	4		2

Table A.15: Python experiment general impression agreement questions. These questions are on a scale of 1 (strongly disagree) and 5 (strongly agree) inclusive.

Alternate tools. In the question, *what tools would you use instead of Git Thermite to retrieve the same information?*, one participant answered that he would use Git Kraken, but the participant is not sure about it. Another participant answered *gitk*, with a question mark. For extra details, see Appendix C.

A.5.4 NASA TLX form

The responses of the NASA TLX form for measuring cognitive load when using Git Thermite to analyze commits are in Table A.16. The responses of the same form, but when using *Monticello textual diff* tool to analyze the commits are in Table A.17. The differences between these two responses are presented in Table A.18. Like in the case of the Pharo experiment, there is a clear tendency that seems to favor Git Thermite in terms of reduced task load, for all of the the aspects. However, because of the very small sample size ($N = 2$), it is not possible to perform a claim on statistic significancy.

	Mental Demand	Physical Demand	Temporal Demand	Performance	Effort	Frustration
Participant 1	9	9	9	2	4	1
Participant 2	6	1	5	11	6	14
Average	7.5	5	7	6.5	5	7.5
Standard Dev	2.12	5.66	2.83	6.63	1.41	9.19

Table A.16: NASA TLX responses with Git Thermite in the Python experiment. Scale from 1 to 21, inclusive. Middle point at 11.

	Mental Demand	Physical Demand	Temporal Demand	Performance	Effort	Frustration
Participant 1	11	11	11	6	10	1
Participant 2	7	1	13	20	6	14
Average	9	6	12	13	8	7.5
Standard Dev	2.83	7.07	1.41	9.90	2.83	9.19

Table A.17: NASA TLX responses with the *GitHub diff* tool in the Python experiment. Scale from 1 to 21, inclusive. Middle point at 11.

	Mental Demand	Physical Demand	Temporal Demand	Performance	Effort	Frustration
Participant 1	-2	-2	-2	-4	-6	0
Participant 2	-1	0	-8	-9	-0	0
Average	-1.5	-1	-5	-6.5	-3	0
Standard Dev	0.71	1.41	4.24	3.54	4.24	0.00
p value	0.205	0.500	0.344	0.234	0.500	NA

Table A.18: NASA TLX Diff responses minus Git Thermite responses. p-value is computed by using a pairwise two-tailed t-Student test.

Appendix B

Git Thermite learning material

This appendix includes a complete copy of the exact learning material used in the Pharo controlled experiment. This is the learning material about Git Thermite that was given for reading to the participants of the controlled experiment.

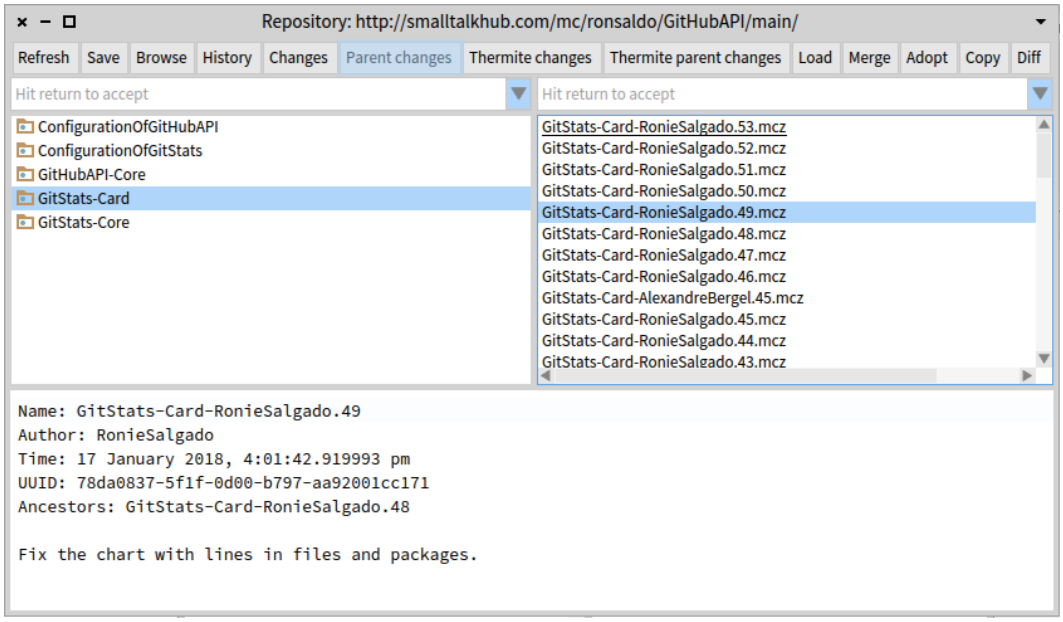
Git Thermite Description

Description of Git Thermite

Git Thermite is a tool to visualize GitHub Pull requests. A pull request in GitHub is a formal request made to the maintainers of an open source project hosted in GitHub to integrate a set of git commits. A pull request can be accepted or rejected by the maintainers of the open source project. Determining whether a pull request deserves to be included in the application mainstream branch is often perceived as a difficult and tedious task since it involves comparing source code. The objective of Git Thermite is to ease this activity by offering a visual tool to project maintainers.

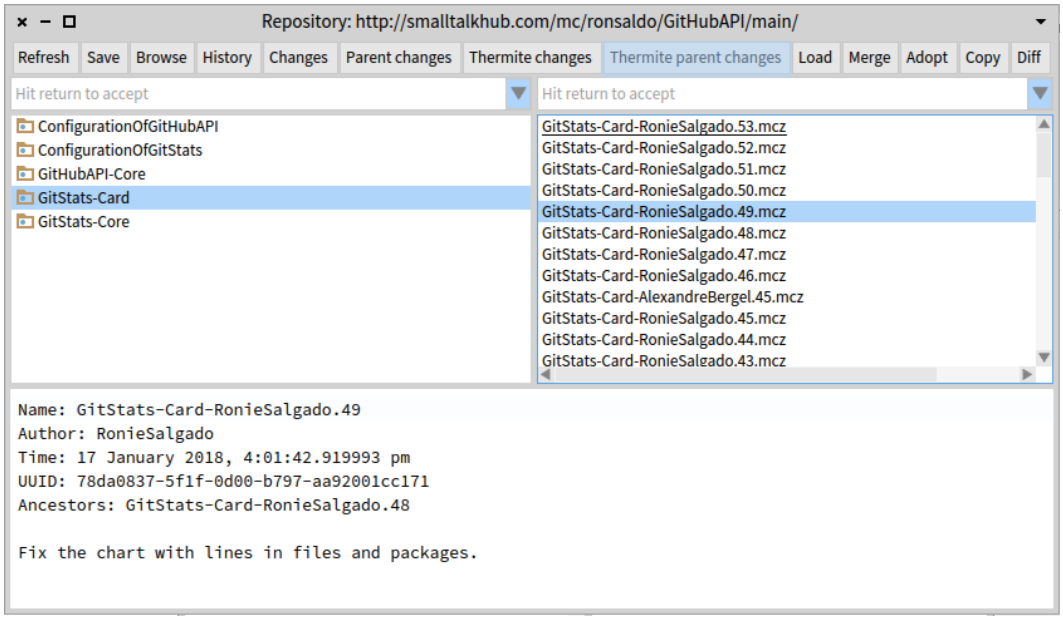
Visualizing a Monticello Commit with Monticello Diff

To visualize a Monticello Commit with the diff tool provided by Monticello, you have to click on *Parent changes* in the Monticello repository browser.



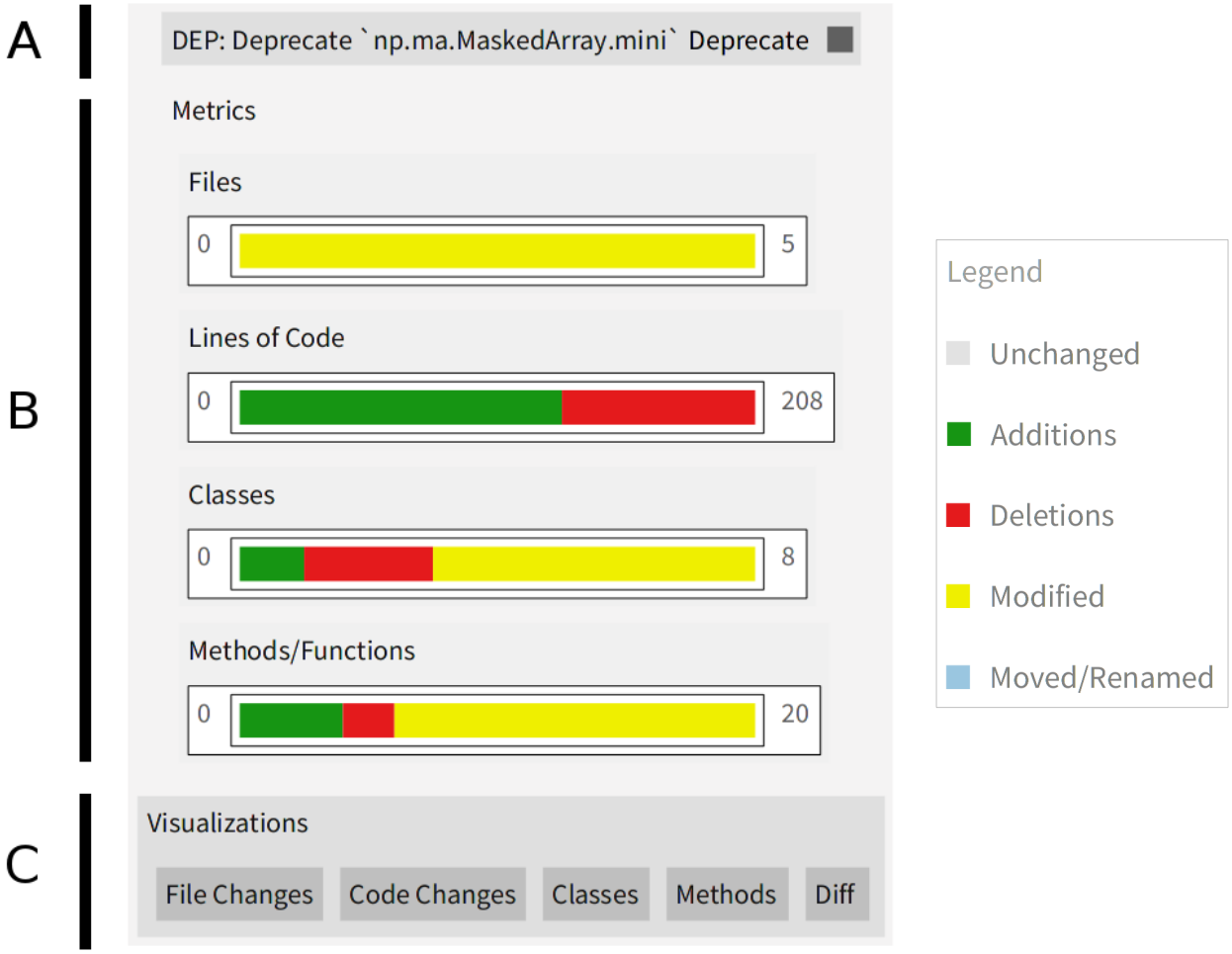
Visualizing a Monticello Commit with Git Thermite

To visualize a Monticello Commit with Git Thermite, you have to click on the *Thermite Parent changes* button in the Monticello repository browser.

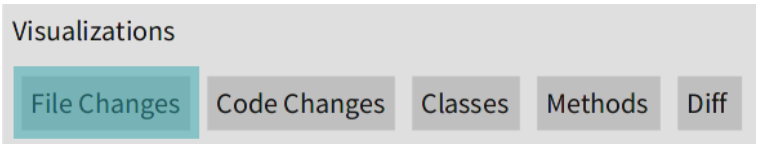


Pull Request Overview

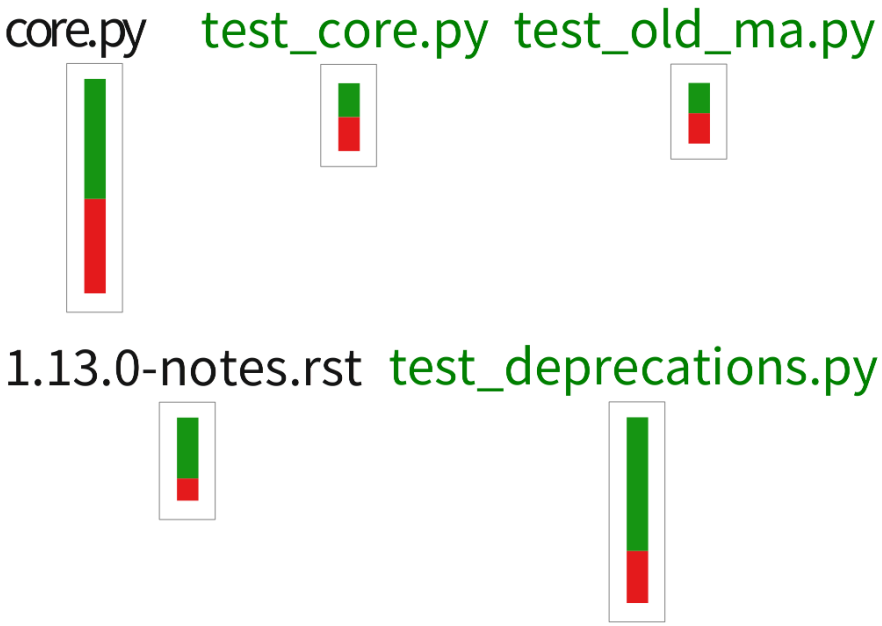
The figure above displays our visualization for a pull request in Python or Pharo. It contains a title bar (Part A) with the name of the pull request, the type of the pull request if it can be inferred, and a square indicating about the possibility of doing an automatic merge. The second section (Part B) displays the different charts with metrics about the modifications done in the pull request. By clicking on the bars of the different charts it is possible obtain a visualization of the elements that are represented by the bar. The last section (Part C) contains a number of navigation buttons to move around into the different visualizations of the pull request provided by Thermite.



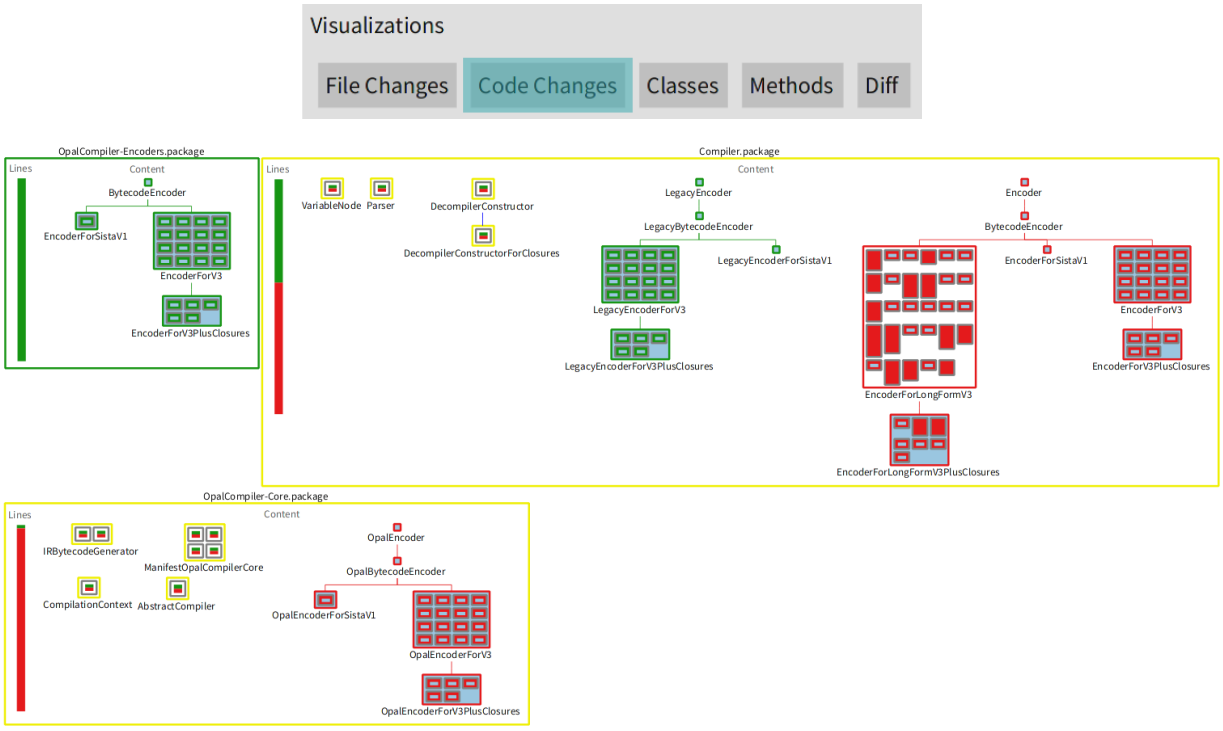
File Changes



The file changes visualization displays changes made to text files in the pull request. In this visualization, each file text is represented by a box with an embedded chart that display the number of lines that are unchanged in gray, added in green and removed in red. If the file name contains the word 'test', then the filename is green to indicates the possibility of having unit test. By hovering the mouse, it is possible to raise a popup with a detailed description of the changes. Clicking on a file opens the text diff with the changes to the clicked file.



Code Changes



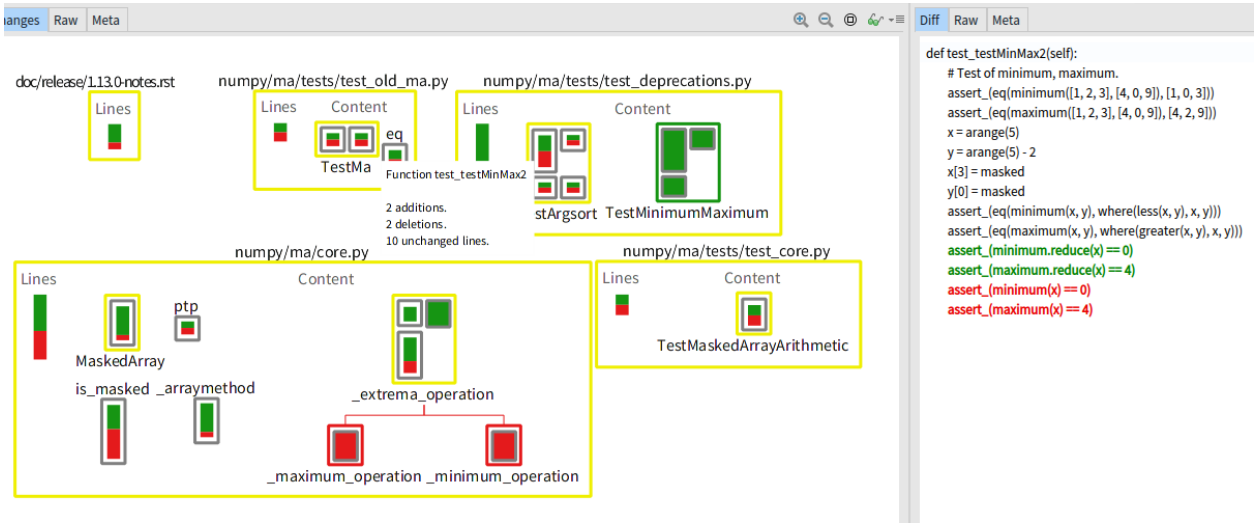
This is the section with the main visualization. This visualization represents all of the structural changes that are done during a pull request or a commit. The structure of the code is represented by the use of embedding for representing the *contains a* relationships between packages, files, classes and methods. The type of change that was made to a container element (package/file/class) is represented by using the border color with the same legend that is used for the metrics in the pull request overview. In the special case of files (Python) and packages (Pharo), there is a chart with the number of lines added/removed/unchanged on the left side of the element.

Methods and function are represented in two different ways: a solid rectangle if the method was completely added/removed/moved or is unchanged; and as a rectangle with an embedded chart with metrics for the number of lines that are unchanged, added or deleted.

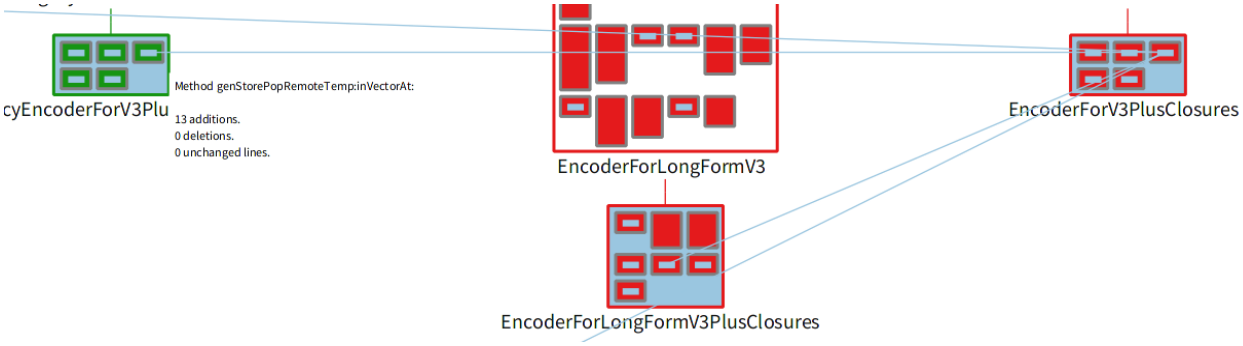
If the options for *Adding unchanged lines/methods/classes* are not selected when building the visualization, these elements are completely omitted from this visualization and all the other ones.

By highlighting an element with the mouse, it is possible to obtain a tooltip with a detailed description of the element, along with its metrics.

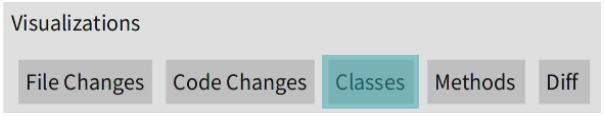
By clicking on an element it is possible to obtain the textual diff of the element. The textual diff of an element uses the green color for added lines, and the red color for lines removed.



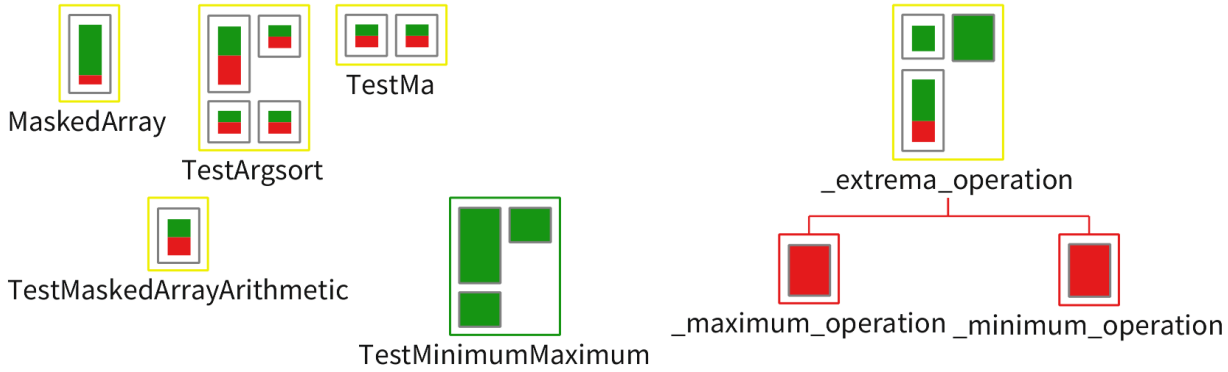
Moved and/or renamed elements are represented with a blue background. Edges connecting the similar elements are displayed when the mouse is highlighting and element that is moved and/or renamed.



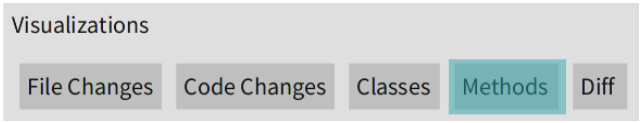
Classes



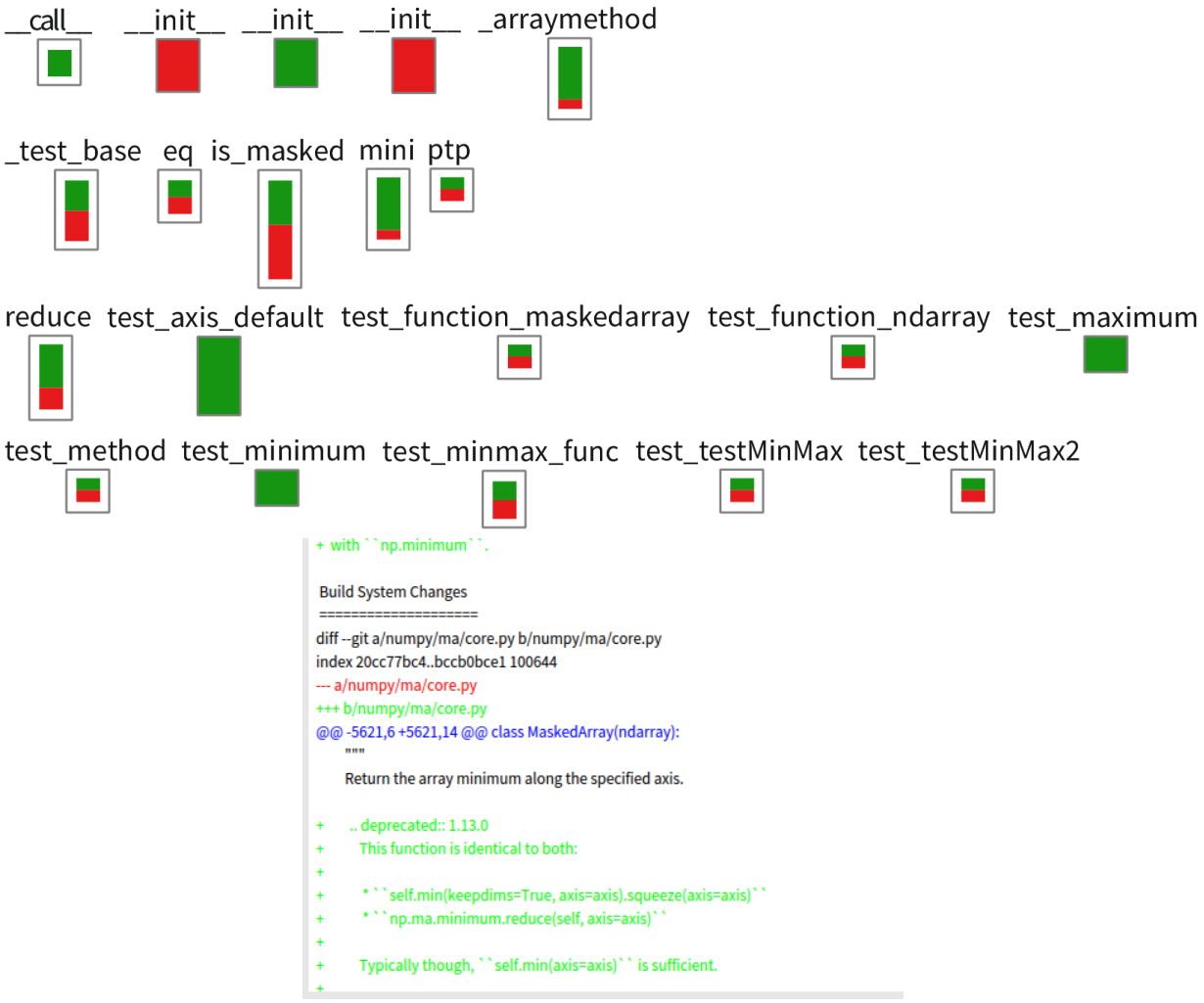
This contains a visualization of the source code changes with only the classes. This visualization does not show packages or the files, which can help on displaying the class hierarchies. By clicking on the bars present in the the chart of number of classes, it is possible to obtain this same visualization for a subset of the classes (e.g: only added classes, or only the removed classes, etc)



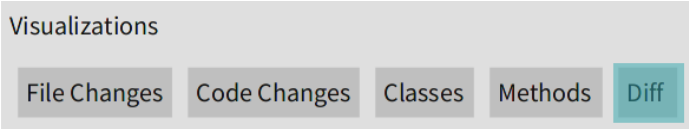
Methods



This contains a visualization with only the changed methods, without the classes and packages.



Diff



This provides a global textual diff with for the whole changes in the pull request.

Appendix C

Controlled experiment exact answer sample

The purpose of this appendix is containing the exact answers given by the participant of the Pharo controlled experiment. Unfortunately, for reasons of space it is not possible to include a printout of this wholedata. Instead we decided to publish the whole data set through Zenodo with DOI: 10.5281/zenodo.2529981, and to provide a scan with the answers of a single participant (Participant 1) in the rest of this appendix. The major treatment to this scan is the anonymization of the NASA TLX form responses through analog mediums.

Git Thermite Controlled Experiment

Date: 21/03/2018
Application: Spy 2 (Familiar) / Roassal (Unfamiliar)

Session overall:

This session aims at doing an experimental study of the *Git Thermite* visualization tool with practitioners.
The session is divided into four phases. The first phase is about yourself. In the second phase you will be given learning material about Git Thermite. The third phase is about answering some questions about the changes introduced during four commits in a provided application. Two of these commits are analyzed by using Monticello Diff, and the other two of these commits are analyzed by using Git Thermite. The fourth phase is about giving general impressions about the tool.
Note that all your answers are treated anonymously.

Phase 1: About yourself

- How many years of experience do you have in programming?
  7 years
- How long have you been programming in Python?
  6 months
- How long have you been programming in Pharo?
  5 years
- Which other languages do you use?
  Javscript
- How old are you?
  26
- What is your gender?
  Male
- What is your highest achieved academic degree?
  Master
- Are you a student or a professional practitioner?
  Profesional
- How familiar are you with using visualizations? Choose one of the following

___ Zero: I have never used a visualization.
___ Low: I almost never use a visualization.
_X_ Medium: I regularly use a visualization.
___ High: I use quite frequently a visualization, or it is part of my daily job.

- Do you use Git?

  Yes
- Which other version control system do you use?

  Monticello

- Have you made a commits in GitHub? if yes, do you use or make commitss on GitHub on a regular basis?

  Yes, long time ago (3years)
- How do you review source code changes?

  Using monticello changes or git-diff

- How do you integrate commits made by a developer other than you?

  Yes

- How do you verify a commit made by a developer other than you?

  Manual source code inspection and tests

- How do you navigate in the history of a project?

  Using gitlab, bitbucket, etc... tools.

- What are the limitations you see in Monticello and/or git diff?

  If the commit is too large it gets very difficult to understand what is happening.
- Do you consider yourself an expert in the application used in this study?

  Spy 2 Yes Roassal 2 Medium
- Who do you think are the experts in the application used in this study?

  Spy 2 Me Roassal 2 Alexandre.

Phase 3: Commit Analysis — Roassal

COMMIT 1: 1619 – 1648

TOOL: Roassal 2 — Git Thermite

TIME BEGIN: 17:38

- Characterize the size and complexity of the commits? (trivial vs complicated)
  Please, justify

  It is a large commit with more than 5000 lines changed. Also it is complex because there are too many changes of different domain. There should split the commit.

- Does the commits actually include unit tests?
  Please, justify

  Yes. New test RTMondrianTest >> testNodesError

- How are the changes in the commits? Are things being mostly added, removed, modified, or something else?
  Please, justify

  Mostly added.

  113 new methods vs 44 modified vs 6 removed.

- Is there a refactoring (class rename, code movement) in the commits?
  Please, justify

  Yes. RTAboutRoassal and RTHowToContributeRoassal are moved to become subclasses of new class RTAbstractWorldMenu

- Can you describe the class hierarchy involved in the pull request / commit?
  Please, justify

  There is a new hierarchy called RTAbstractWorldMenu. Also there are changes in RTAbstractRegularTreeLayout.

- Do you find easy to understand the changes in the commits? *(strongly agree, agree, neutral, disagree, strongly disagree)*
  Please, justify

  Neutral.
  Some of them are easy to understand, mostly the most general ones (the overview).
  The relation between changes is not easy unless they are related by the class hierarchy.

TIME END: 17:50

COMMIT 2: Roassal 2

TOOL: Monticello diff

TIME BEGIN: 17:51.

- Characterize the size and complexity of the commits? (trivial vs complicated)
  Please, justify

  It looks like a medium size commit with lots of refactorings and renaming of methods.

- Does the commits actually include unit tests?
  Please, justify

  Yes. But not many.
  RTMondrianTest >> testSourceError.

- How are the changes in the commits? Are things being mostly added, removed, modified, or something else?
  Please, justify

  I see many methods being renamed or refactored. This is translated to many additions and removals.

- Is there a refactoring (class rename, code movement) in the commits?
  Please, justify

  Yes! Some ClassDescription methods are moved to TClassDescription.
  Also RTForceLayout has been declared obsolet and new RTForceLayoutStepping is being implemented.

- Can you describe the class hierarchy involved in the pull request / commit?
  Please, justify

  There are members of layout hierarchy such as RTAbstractRegularTree layout and RTForceL...
  Also members of Examples hierarchy such as RTExperimentalExample.

- Do you find easy to understand the changes in the commits? *(strongly agree, agree, neutral, disagree, strongly disagree)*
  Please, justify

  Disagree. It is very easy to get lost reviewing the changes, and there is a high chance that I have missed things.

TIME END: 18:04

COMMIT 3:

TOOL: Thermite

TIME BEGIN: 18:02

- Characterize the size and complexity of the commits? (trivial vs complicated)
  Please, justify

  It is mostly a simple refactoring introducing a superclass to Tests. Also there seem to be other unrelated features and removals performed.
- Does the commits actually include unit tests?
  Please, justify

  Yes. RTViewTest >> testRemovingEdges.

- How are the changes in the commits? Are things being mostly added, removed, modified, or something else?
  Please, justify

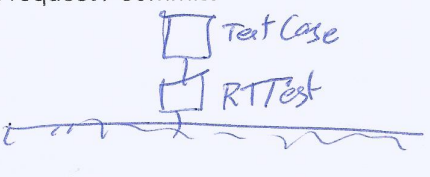
  Regarding the classes, there was a massive refactoring on the test hierarchies. Lots of modified superclass.
  Also some classes (?) with several methods where removed, so also some methods removal. (?).
- Is there a refactoring (class rename, code movement) in the commits?
  Please, justify

  Yes! The whole test hierarchy has been changed introducing a new common superclass for Roassal tests called RTTest.

- Can you describe the class hierarchy involved in the pull request / commit?
  Please, justify

  RTTest hierarchy is very flat.

  Single layer of many classes.



- Do you find easy to understand the changes in the commits? *(strongly agree, agree, neutral, disagree, strongly disagree)*
  Please, justify

  Agree. The addition of this RTTest is easy to spot.

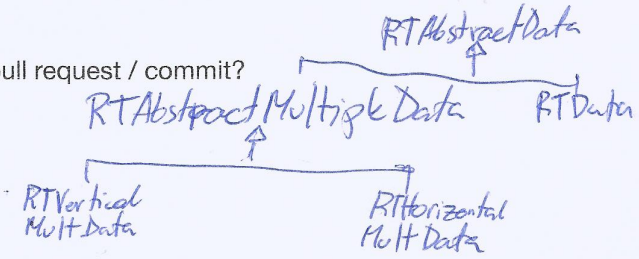
TIME END: 18:12.

COMMIT 4:

TOOL: Monticello

TIME BEGIN: 18:13.

- Characterize the size and complexity of the commits? (trivial vs complicated)
  Please, justify
  It is a rather small commit (compared to the previous ones). Also it looks like the main thing was a refactoring of RT Multiple Data.
- Does the commits actually include unit tests?
  Please, justify
  Yes. RT Vertical Multiple Data Test >> test Basic.
- How are the changes in the commits? Are things being mostly added, removed, modified, or something else?
  Please, justify
  Mostly added but also a refactoring that splitted a large class in smaller classes. Lots of adding and some removals.
- Is there a refactoring (class rename, code movement) in the commits?
  Please, justify
  Yes. The splitting of RT Multiple Data.
- Can you describe the class hierarchy involved in the pull request / commit?
  Please, justify
  RT Multiple Data ⟹



- Do you find easy to understand the changes in the commits? *(strongly agree, agree, neutral, disagree, strongly disagree)*
  Please, justify
  Agree, though it would be nice to be able to see the class hierarchy to look how far the RT Multiple Data went into the RT Abstract Data Hierarchy.

TIME END: 18:20.

Phase 4: General impressions

TIME BEGIN: 18:21

QUESTION 1:
What do you think about the content visualization?

- I find it easy to understand in general:
  (strongly agree, agree, neutral, disagree, strongly disagree)
  Please, justify
  Strongly Agree. Everything is clear and intuitive.

- I find it easy to understand the visual variable mapping:
  (strongly agree, agree, neutral, disagree, strongly disagree)
  Please, justify
  Strongly Agree. The mapping is clear.

- I find it easy to understand the interactions:
  (strongly agree, agree, neutral, disagree, strongly disagree)
  Please, justify
  Strongly Agree. The click action is straightforward and the buttons are easy to understand.

- I find it easy to navigate between visualizations:
  (strongly agree, agree, neutral, disagree, strongly disagree)
  Please, justify
  Strongly Agree. Click + Glamours tabs (views ctrl+g) is easy.

- I find it useful for solving requested tasks:
  (strongly agree, agree, neutral, disagree, strongly disagree)
  Please, justify
  Agree. For many of the refactorings the visualization is usefull to get the overview.

- I find it useful for my daily job:
  (strongly agree, agree, neutral, disagree, strongly disagree)
  Please, justify
  Strongly Disagree.

  In my daily life I do not use such large commits or branches. Also it is a small team so it is easier to review the commits.

  I suggest to use this tool for larger teams.

QUESTION 2:
In what scenarios and for solving which maintenance tasks would developers use Git Thermite?

Please, justify

For larger teams that have long life branches.
If the team is up-to-date with what everybody is doing AND the commits (pull requests) are small, there is no need of using a more powerfull tool such as thermite.

QUESTION 5:
What tools would you use instead of Git Thermite to retrieve the same information?

Please, justify

Paassal, regular expressions and a headache.

QUESTION 6:
Do you have any suggestion on how to improve the visual report or the tool itself? Any critic?

Please, justify

Just to remove the 0 in the scale.

TIME END: 18:31

Grammar Code - Spy 12/04/2018

Phase 3: Commit Analysis

COMMIT 1: 30-13

TOOL: Thermite

TIME BEGIN: 17:12

- Characterize the size and complexity of the commits? (trivial vs complicated)
  Please, justify

  Sequence of small commits. The biggest change is the implementation of some concurrent behaviors like ThreadSafeDict.

- Does the commits actually include unit tests?
  Please, justify

  No, the commits do not have test. Though, the test are probably included in Spy 2-Core-Tests.

- How are the changes in the commits? Are things being mostly added, removed, modified, or something else?
  Please, justify

  Mostly addition of code.

- Is there a refactoring (class rename, code movement) in the commits?
  Please, justify

  No, it seems it is just adding features and bug fixing.

- Can you describe the class hierarchy involved in the pull request / commit?
  Please, justify

  There is not any important class hierarchy involved.
  Spy2 hierarchy is just for adding utility methods, test methods and supporting singleton.

- Do you find easy to understand the changes in the commits? *(strongly agree, agree, neutral, disagree, strongly disagree)*
  Please, justify

  Strongly Agree.
  I can easily identify the new classes being added, which correspond to the new features and, in this case, the code modification for bug fixing.

TIME END: 17:22.

COMMIT 2: 22-29

TOOL: Dibb

TIME BEGIN: 17:22

- Characterize the size and complexity of the commits? (trivial vs complicated)
  Please, justify

  There are 4 different changes in the analyzed pull request. 2 bug-fixing
  1 cleaning
  1 refactoring

- Does the commits actually include unit tests?
  Please, justify

  No, but probably present in "SpZ-Core-Tests.

- How are the changes in the commits? Are things being mostly added, removed, modified, or something else?
  Please, justify

  Mostly changes

- Is there a refactoring (class rename, code movement) in the commits?
  Please, justify

  Yes. S2Profiler>>runUnitTests now use runTestWithResult making the code cleaner.

- Can you describe the class hierarchy involved in the pull request / commit?
  Please, justify

  No.
  SpyZ - S2Profiler } Not important for this case.
  S2Class ...

- Do you find easy to understand the changes in the commits? *(strongly agree, agree, neutral, disagree, strongly disagree)*
  Please, justify

  Strongly Agree, changes are pretty straightforward.

TIME END: 17:32.

COMMIT 3: 18-21

TOOL: Thermite

TIME BEGIN: 17:32

- Characterize the size and complexity of the commits? (trivial vs complicated)
  Please, justify

It is a small commit. Just 4 classes changed.

- Does the commits actually include unit tests?
  Please, justify

No, but probably included in Spy2-Core-Tests.

- How are the changes in the commits? Are things being mostly added, removed, modified, or something else?
  Please, justify

Created a new class and moved methods from S2AbstractInstrumentator to S2Gatherer.

- Is there a refactoring (class rename, code movement) in the commits?
  Please, justify

Yes! The whole is a refactoring about splitting the responsibilities of S2AbstractInstrumentator.

- Can you describe the class hierarchy involved in the pull request / commit?
  Please, justify

Yes! The S2AbstractInstrumentator hierarchy.

- Do you find easy to understand the changes in the commits? *(strongly agree, agree, neutral, disagree, strongly disagree)*
  Please, justify

Strongly Agree. I recall what changes I performed and why.

TIME END: 17:40.

COMMIT 4: 11-17

TOOL: Diff

TIME BEGIN: 17:40

- Characterize the size and complexity of the commits? (trivial vs complicated)
  Please, justify

Medium size commit. Cleaning of SZMethod and Implementation of SZWeakLargeIdentityDictionary. Also bugfix about traits.

- Does the commits actually include unit tests?
  Please, justify

No. Probably in ~~SZ~~ SpZ-Core-Tests.

- How are the changes in the commits? Are things being mostly added, removed, modified, or something else?
  Please, justify

In the cleaning of SZMethod → Mostly removes.
SZWeakIdentityDictionary → Mostly additions.
Trait bugfix → Additions and modifications.

- Is there a refactoring (class rename, code movement) in the commits?
  Please, justify

No.

- Can you describe the class hierarchy involved in the pull request / commit?
  Please, justify

No, even though SZAbstractInstrumentator is superclass of SZInstrumentator, the method involved is not used or ~~the~~ implemented by other members of the hierarchy.

- Do you find easy to understand the changes in the commits? *(strongly agree, agree, neutral, disagree, strongly disagree)*
  Please, justify

Strongly agree.
The cleaning of SZMethod, the bug-fix of traits and the implementation of SZWeakLargeIdentityDict are easy to understand.

TIME END: 17:54.

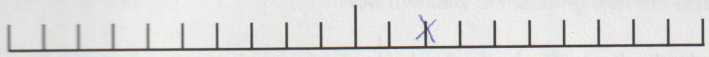
Figure 8.6

NASA Task Load Index

Hart and Staveland's NASA Task Load Index (TLX) method assesses work load on five 7-point scales. Increments of high, medium and low estimates for each point result in 21 gradations on the scales.

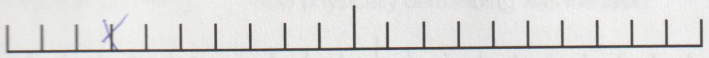
PARTICIPANT | Task: Therm:t | Date: 21/03/2018

Mental Demand — How mentally demanding was the task?



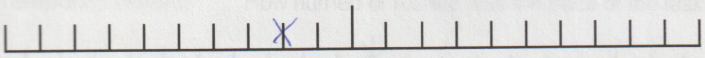
Very Low — Very High

Physical Demand — How physically demanding was the task?



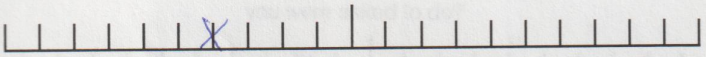
Very Low — Very High

Temporal Demand — How hurried or rushed was the pace of the task?



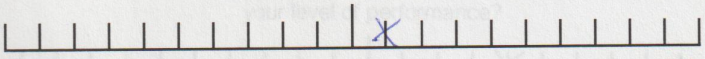
Very Low — Very High

Performance — How successful were you in accomplishing what you were asked to do?



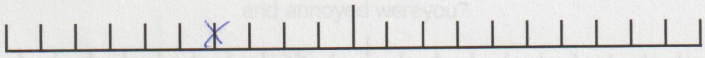
Perfect — Failure

Effort — How hard did you have to work to accomplish your level of performance?



Very Low — Very High

Frustration — How insecure, discouraged, irritated, stressed, and annoyed wereyou?



Very Low — Very High

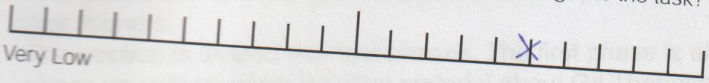
Figure 8.6

NASA Task Load Index

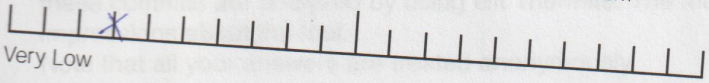
Hart and Staveland's NASA Task Load Index (TLX) method assesses work load on five 7-point scales. Increments of high, medium and low estimates for each point result in 21 gradations on the scales.

PARTICIPANT1 | Task: Diff | Date: 21/03/2018

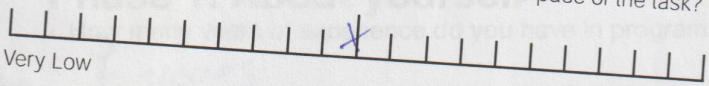
Mental Demand — How mentally demanding was the task?



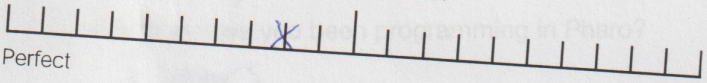
Physical Demand — How physically demanding was the task?



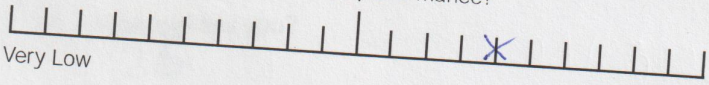
Temporal Demand — How hurried or rushed was the pace of the task?



Performance — How successful were you in accomplishing what you were asked to do?



Effort — How hard did you have to work to accomplish your level of performance?



Frustration — How insecure, discouraged, irritated, stressed, and annoyed wereyou?