



HAL
open science

Supporting Software Integration Activities with First-Class Code Changes

Martín Dias

► **To cite this version:**

Martín Dias. Supporting Software Integration Activities with First-Class Code Changes . Programming Languages [cs.PL]. Laboratoire d'Informatique Fondamentale de Lille, 2015. English. NNT : . tel-01247696

HAL Id: tel-01247696

<https://inria.hal.science/tel-01247696>

Submitted on 22 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Supporting Software Integration Activities with First-Class Code Changes

THÈSE

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille
(spécialité informatique)

par

Martín Dias


Composition du jury

Président : Alain PLANTEC
Rapporteur : Tom MENS, Andreas ZELLER
Examineur : Alain PLANTEC
Directeur de thèse : Stéphane DUCASSE
Co-Encadrant de thèse : Damien CASSOU

Laboratoire d'Informatique Fondamentale de Lille — UMR CNRS 9189
INRIA Lille - Nord Europe
Numéro d'ordre : 41872

Copyright © 2015 by Martín Dias

RMoD
Inria Lille – Nord Europe
Parc Scientifique de la Haute Borne
40, avenue Halley
59650 Villeneuve d'Ascq
France
<http://rmod.inria.fr/>

 This work is licensed under a *Creative Commons Attribution-ShareAlike 4.0 International License*.

Acknowledgments

I wish to thank first to my supervisors Stéphane and Damien, from whom I learnt valuable things during this work. I deeply appreciate your support, encouragement and advice. Many thanks to Stef, Inria and the Nord-Pas de Calais region for giving me the opportunity to experience this great three-year adventure in Lille, which included doing fascinating work and making awesome friends (no need to mention your names).

I thank the RMoD team in general for sharing discussions, efforts, and nice moments everyday. For example, the support from Nicolas Anquetil and Anne Etien. I am also very grateful other Inria members such as Matias Martinez. The Pharo and Smalltalk communities have really cool people too, such as Max Leske.

Respecting this manuscript and my defense, I'd like to express my gratitude to Tom Mens, Andreas Zeller, and Alain Plantec for being part of my Ph.D. committee. I was honored to count with your presence and appreciated your feedback. Also, I'd like to thank Anne and Vincent for their help in the translation of the abstract of this thesis. Special thanks also to Matias, Anne, Damien, Pablo, Santiago and Nata to prepare the defense.

I thank to researchers involved in collaboration works: Guille Polito, Nicolas Anquetil, Verónica Uquillas-Gomez, Alberto Bacchelli and Georgios Gousios.

I am indebted to the public education systems in Argentina and France which allowed meeting amazing persons and being a more complex and happier person.

Finally, thanks to Nataly, who was a key support along these years, and to my family in Burzaco.

Abstract

Software codebases change for many reasons: for example, domain rules might evolve and codebase dependencies, such as used libraries, might be updated. As Lehman's laws suggest, software must continuously change to be useful, often leading systems to grow in size and complexity. Several developers collaborate in the evolution of a complex system. Developers typically change codebases in parallel from each other, which results in diverging codebases. Such diverging codebases must be integrated when finished.

Integrating diverging codebases involves difficult activities. For example, two changes that are correct independently can introduce subtle bugs when integrated together. Integration can be difficult with existing tools, which, instead of dealing with the evolution of the actual program entities being changed, handle code changes as lines of text in files.

Tools are important: software development tools have greatly improved from generic text editors to *integrated development environments* (IDEs) by providing high-level code manipulation such as automatic refactorings and code completion. This improvement was possible by the *reification* of program entities, *i.e.*, by their explicit modeling as first-class entities. Nevertheless, whereas the reification of program entities in IDEs improved productivity in programming practice, integration tools miss reified change entities to improve productivity in integration.

In this work, we study the activities involved in codebase integration, and propose approaches to support integration. First, we conducted an exploratory study to understand what are the most relevant problems in integration activities that have little tool support. We used such information as guidelines to propose:

- EPICEA, a first-class change model and associated IDE tools. EPICEA model includes: *low-level code changes* such as *class addition* and *method modification*; *high-level code changes* such as the *method rename* refactoring; and *IDE interaction data* such as *unit-test run*.
- EPICEAUNTANGLER, an approach to help developers share untangled commits (aka. atomic commits) by using fine-grained code change information gathered from the IDE through EPICEA model and tools.

EPICEA model and tools allowed evaluation of our thesis in real-world scenarios. EPICEAUNTANGLER's results showed that three EPICEA features are especially important to perform clustering of fine-grained code changes: the time between the changes; the number of other modifications between the changes; and whether the changes modify the same class. EPICEAUNTANGLER is based and tested on a large EPICEA dataset that we make publicly available.

Keywords: Software evolution, first-class code changes, integration activities, untangling changes.

Résumé

Le code source des logiciels change pour de nombreuses raisons: par exemple, les règles du domaine peuvent évoluer, ou les dépendances entre les différentes parties du système, telles les bibliothèques, peuvent être mises à jour. Comme les lois de Lehman le suggèrent, le logiciel doit changer pour être en permanence opérationnel, souvent conduisant les grands systèmes à croître en taille et en complexité. Plusieurs développeurs collaborent dans l'évolution d'un système complexe. Les développeurs changent généralement le code source en parallèle des uns des autres, ce qui entraîne des divergences dans le code. Ces divergences se doivent d'être fusionnées.

L'intégration de code source divergent est une activité complexe. Par exemple, deux changements qui sont corrects indépendamment peuvent introduire des bugs subtils lorsqu'ils sont intégrés ensemble. L'intégration peut être difficile avec les outils existants, qui, au lieu de faire face à l'évolution des entités réelles du programme modifié, gère les changements de code au niveau des lignes de texte dans des fichiers sources.

L'outillage est importants: les outils de développement de logiciels se sont grandement améliorés en partant d'éditeurs de texte génériques à des *environnements de développement intégrés* (IDE), qui fournissent de la manipulation de code de haut niveau tels que la refactorisation automatique et la complétion de code. Cette amélioration a été possible grâce à la *réification* des entités de programme, à savoir, leur modélisation explicite comme des entités de première classe. Néanmoins, alors que la réification des entités de programme dans les IDEs ont amélioré la productivité dans la pratique de la programmation, les entités de changement réifiées manquent dans les outils pour améliorer la productivité dans l'intégration.

Dans ce travail, nous étudions les activités impliquées dans l'intégration de code source, et proposons des approches pour faciliter l'intégration. Tout d'abord, nous avons mené une étude exploratoire pour comprendre quels sont les problèmes d'outillage les plus pertinents dans les activités d'intégration. Nous avons utilisé de telles informations comme lignes directrices pour proposer:

- EPICEA, un modèle de changement de première classe et des outils d'IDE associés. Le modèle d'EPICEA comprend: *des changements de code de bas niveau* tels que l'ajout de classe et la modification de méthodes; *des modifications du code de haut niveau* tels que le refactoring de changement de nom de méthode; et *des données d'interaction avec l'IDE* tels que l'exécution de tests unitaires.
- EPICEAUNTANGLER, une approche pour aider les développeurs à démêler les commits (par une décomposition en commits atomiques) en se basant sur des changements de code à grain fin recueillis auprès de l'IDE grâce au modèle et aux outils d'EPICEA.

Le modèle et les outils d'EPICEA permettent l'évaluation de notre thèse dans les scénarios du monde réel. Les résultats d'EPICEAUNTANGLER ont montré que trois caractéristiques d'EPICEA sont particulièrement importantes pour regrouper les changements de code à grain fin: la durée entre les changements; le nombre d'autres modifications entre les changements; et si les changements modifient la

même classe. EPICEAUNTANGLER est basé et testé sur un grand ensemble de données provenant d'EPICEA que nous mettons à la disposition du public.

Mot clés: évolution de logiciels, réification des changement de code, intégration de branches, démêlage de changements de code.

Contents

1	Introduction	1
1.1	Integration Activities	1
1.2	Integration Tools	2
1.3	Thesis	2
1.4	Contributions and Roadmap	3
1.5	The Case of PHARO/SMALLTALK	5
2	Integration Activities	7
2.1	Related Work	8
2.2	Methodology	11
2.3	Catalog of Integrator’s Questions	12
2.4	Results	15
2.4.1	Participant and System Profiles	15
2.4.2	Integrator’s Questions	18
2.4.3	Threats to Validity	18
2.5	Discussion: Top Important Questions without Tool Support	20
2.6	Conclusion	23
3	First-Class Code Changes	25
3.1	Version Control Systems	26
3.2	Reconstructing First-class Evolution	27
3.3	Recording First-Class Evolution	28
3.4	Change Management in PHARO	29
3.5	EPICEA	31
3.6	EPICEA Model	31
3.6.1	Code Change Model	32
3.6.2	IDE Event Model	33
3.7	EPICEA Tools	34
3.8	Conclusion	36
4	Untangling Code Changes	37
4.1	Problem Description	39
4.1.1	Existing Solutions for Tangled Changes	39
4.1.2	Addressing the Current Limitations	41
4.2	Proposed Solution: EPICEAUNTANGLER	42
4.2.1	Gathering Fine-Grained Changes and IDE Events	42
4.2.2	Voters	42
4.2.3	Machine Learning Approaches	46
4.2.4	Clustering	46
4.3	Research Method	47
4.3.1	Research Questions	47
4.3.2	Research Settings	48
4.3.3	Research Steps	48
4.4	Results	53

4.4.1	What Are the Dominant and Significant Voters?	53
4.4.2	How Effective Is Random Forests with the Dominant Voters?	54
4.4.3	How Effective Is EPICEAUNTANGLER for Developers?	57
4.5	Discussion	58
4.5.1	Results	58
4.5.2	Threats to Validity	59
4.6	Related Works	60
4.7	Conclusion	62
5	Conclusion and Future Work	63
5.1	Summary	63
5.2	Future Work	65
5.2.1	Interaction Data to Improve Modern Code Review	66
5.2.2	Interaction Data to Mitigate Ripple Effects	67
5.3	Publications	69
	Bibliography	73
A	Appendix	83
A.1	PHARO Programming Language	83
A.1.1	Minimal Syntax	84
A.1.2	Message Sending	84
A.1.3	Precedence	85
A.1.4	Cascading Messages	86
A.1.5	Blocks	86
A.1.6	Methods	87

1 INTRODUCTION

Contents

1.1	Integration Activities	1
1.2	Integration Tools	2
1.3	Thesis	2
1.4	Contributions and Roadmap	3
1.5	The Case of PHARO/SMALLTALK	5

Software systems are indispensable in today's world. Computing devices are everywhere, and continue gaining in popularity. This increases the challenges of software development [Mens 2005, v. Deursen 2008], *e.g.*, evolving domains that require program adaptation, new technologies and libraries that require program migration, etc. As Lehman's laws suggest [Lehman 1980], software must be continuously adapted to be useful. Software is in constant evolution, leading systems to grow in size and complexity. Behind a complex system, numerous developers collaborate in the evolution of the system's codebase. Due to this complexity of software evolution, developers need as much help as they can get from tools.

During its lifetime, a system's codebase changes due to concerns such as bug fixes, new features, and migrations [Demeyer 2002]. According to best practices, the implementation of such concerns happen in isolation from each others, to avoid potential interferences while they are not ready. This means that codebases temporarily diverge during development and must be integrated back when finished.

1.1 Integration Activities

In the development team of a software system, the *integrators* have the crucial role of integrating developers' code changes. Also known as project maintainers or integration managers, their work involve diverse *integration activities* such as preventing introduction of bugs, keeping code conventions, avoiding code duplication, and keeping architectural decisions.

For each code change Δ , integrators ask themselves questions such as:

- What program entities (*e.g.*, packages, classes, methods) does Δ change?
- Is the content of Δ cohesive? Or should it be split?
- Will client systems need to change their codebases because of Δ ?
- Have program entities related to Δ suffer other changes since the codebase diverged?

Those questions illustrate some inherent difficulties in integration activities. They involve understanding somebody else's changes, to be integrated in a code-base that is often different than the original one where the developer worked.

1.2 Integration Tools

At the technical level, software developers manage the evolution of a system code-base in a *version control system* (VCS) such as GIT, MERCURIAL, SVN or MONTICELLO. Most VCS version files and directories. Basic versioning operations are *commit* (store or check-in) files, *checkout* (restore or clone) files, or *diff* (compare) file versions. Other fundamental operations are: *branch*, which creates a copy of the versioned files, allowing isolated changes; and *merge*, which reconciles multiple branches.

VCS merging tools implement semi-automatic algorithms to facilitate integration, which make optimistic assumptions about the internal structure of the files. In some cases, an automatic merge can be successfully performed, while in other cases, a person must decide exactly what the resulting files should contain.

In any case, current merging tools manage the evolution of files instead of program entities, which can lead to either wrong automatic tool decisions or wrong integrator decisions. We argue that the semantic gap between the program entities (reality) and the files (representation) is substantial, hindering integration activities and, by consequence, the whole software development process.

1.3 Thesis

Taking a look at the history of software development tools, we find that they have greatly improved from generic text editors to *integrated development environments* (IDEs). Modern IDEs use *abstract syntax trees* (ASTs) to provide high-level code manipulation (such as automatic refactorings [Fowler 1999b]), only possible by the *reification* of the domain entities, *i.e.*, by the explicit modeling of first-class program entities.

However, while the reification of domain entities as first-class objects in IDEs improved developers' productivity, modern VCS tools still miss reifying their domain entities to improve integrators' productivity.

In short. *We claim that the reification of domain entities of software evolution provides a more comprehensive support to integration activities.*

1.4 Contributions and Roadmap

Following, we state the main contributions of this work together with a pointer to the corresponding chapter.

Key integration activities without comprehensive tool support. Since we orient our work to tackle real-world integration problems, we needed to know the key (most relevant) integration activities without comprehensive tool support. However, we found that there is no quantitative information in that regard in literature. Thus, we performed an exploratory study to discover which are the key integration activities and the level of tool support for each. (In **Chapter 2**.)

Requirements for a model of software evolution. We want to narrow the semantic gap present in nowadays' management of software evolution. Then, we present the state-of-the-art in modeling software evolution and mark limitations. We conclude that such limitations can be better addressed with a first-class change model and developer interaction data gathered from the IDE when working. (In **Chapter 3**.)

EPICEA model. We report on a model of software evolution which includes explicit entities for:

- *low-level code changes*, such as *class addition* and *method modification*;
- *high-level code changes*, such as *method rename* refactoring;
- *IDE interaction data*, including extra developer's actions in the IDE such as *unit test run*.

We propose this model as a means to overcome limitations of traditional software engineering approaches which mine software repositories to reconstruct the history of a software. (In **Chapter 3**.)

EPICEA monitor and associated tools. The EPICEA change model requires information that is absent in VCS, hindering reconstruction from VCS data. However, such information can be gathered from the developer when working. Thus, we implemented an EPICEA monitor, that listens developer actions from the IDE and records them as EPICEA model instances. We report on the implementation of EPICEA monitor and other associated tools, that allow us to evaluate our approaches in realistic scenarios. (In **Chapter 3**.)

EPICEA UNTANGLER. After working for some time, developers commit their code changes to a VCS. When doing so, they often bundle unrelated changes (*e.g.*, bug fix and refactoring) in a single commit, thus creating a so-called tangled commit.

Sharing tangled commits is problematic because it makes review, reversion, and integration of these commits harder and historical analyses of the project less reliable. We report on a novel approach, `EPICEAUNTANGLER`, to help developers share untangled commits (aka. atomic commits) by using fine-grained code change information gathered from the IDE through `EPICEA` model and tools. (In **Chapter 4**.)

`EPICEAUNTANGLER` public dataset. `EPICEAUNTANGLER` is based and tested on a publicly available dataset. This dataset of untangled code changes was created with the help of two developers who accurately split their code changes into self contained tasks over a period of four months. (In **Chapter 4**.)

`EPICEAUNTANGLER` evaluation. We evaluated `EPICEAUNTANGLER` by deploying it to 7 developers, who used it for 2 weeks. We recorded a median success rate of 91% and average one of 75%, in automatically creating clusters of untangled fine-grained code changes. (In **Chapter 4**.)

In **Chapter 5** we conclude this work by summarizing and discussing our work. Additionally, we outline future research plans using `EPICEA` model of developer's activities in the IDE (*i.e.*, *interaction data*). We draw up research tracks using interaction data in two case studies: *improve modern code review* and *mitigate change ripple effects* (external impact).

1.5 The Case of PHARO/SMALLTALK

As stated above, integration of code changes involves hard activities that miss comprehensive tool support. In particular, we point that VCS integration tools treat code changes as lines of text in files instead of dealing with the evolution of actual programs and the entities that constitute them. Then, we propose to improve integration tools with semantic awareness of the program entities involved in the changes.

Our work is empirical in nature, since we focus on real-world integration problems. For this work we chose the *Pharo project* [Black 2009]¹ as a case study. PHARO is a dynamic language based on SMALLTALK [Goldberg 1989] with its own open-source programming environment.² PHARO not only illustrates the integration problems but also serves us as a source of professional developers, researchers, and students to evaluate our approaches. This provides us with real feedback from developers that actually face the inherent problems of branching and merging.

PHARO is used by more than 25 universities³ worldwide to teach programming, by 15 research groups to build tools, and more than 50 companies are using it in production⁴.

Why PHARO? There are two main motivations to use PHARO as a case study for our work on integration activities. First, the versioning system is tightly integrated in the IDE and already provides some first-class capabilities. Second, an important aspect in our decision is that the PHARO community of developers has been receptive, since its inception, to welcome and thoroughly evaluate research tools [Renggli 2010, Verwaest 2011, Hora 2014, Uquillas Gómez 2012b].

It might be considered a drawback to choose a dynamically-typed language whereas most research approaches rely on static analysis of programs that are statically-typed (JAVA, C#, etc.). Certain types of static analysis, *e.g.*, accurate call graph analysis, is not possible for dynamically-typed languages (JAVASCRIPT, RUBY, PYTHON, etc.). Therefore, our approaches cannot strongly rely on such static analysis. However, the broad use of dynamic-languages nowadays makes worth to explore this less-explored setting.

¹PHARO: <http://pharo.org/>

²An overview of the PHARO language is provided in Appendix A.

³List of universities teaching PHARO: <http://pharo.org/Teachers>

⁴List of companies using PHARO: <http://pharo.org/success>

2 INTEGRATION ACTIVITIES

Contents

2.1	Related Work	8
2.2	Methodology	11
2.3	Catalog of Integrator's Questions	12
2.4	Results	15
2.5	Discussion: Top Important Questions without Tool Support	20
2.6	Conclusion	23

Introduction

Software is in constant evolution [Demeyer 2002] [Lehman 1980]. In a software project, code changes represent bug fixes, enhancements, new features and adaptations due to changing domains. The evolution of a project codebase is usually managed in a VCS supporting branches. Developers perform code changes in a branch and often such changes should be integrated into another branch. Integration of changes is a difficult activity and poses substantial challenges [Uquillas Gómez 2012a, Gousios 2014]. Focused on understanding development challenges, several research works [LaToza 2010, Premraj 2011, Sillito 2008, Fritz 2010] systematically characterize what questions developers need to answer when working. These works present catalogs (*i.e.*, lists) of questions that serve as a basis for research on new tools to improve the workflow of developers. Besides the results are useful to understand development activities, the results specifically focused on integration activities are scarce. A recent survey [Gousios 2014] proposes some questions to characterize GITHUB's pull requests. However the authors do not focus on a systematic characterization of questions that integrators ask themselves.

The main contributions of this chapter are:

1. A catalog of 46 questions that integrators ask when performing integration of changes. The main motivation behind obtaining these questions is to identify and understand which are the information needs and tool support of developers that deal with integration activities (Section 2.3).
2. An evaluation of each question of this catalog. For each question, the participants had to rank the importance and the support that tools offer. In a period of 5 months we received the answers of 42 integrators who integrate changes on diverse software projects (Section 2.4).

3. An analysis of the evaluation results, where we identified three key integration activities without comprehensive tool support. These results serve as guidelines to focus efforts on new approaches to improve everyday's work of integrators (Section 2.5).

2.1 Related Work

Questions about Code. LaToza and Myers conducted a survey to investigate the questions that developers consider *hard-to-answer* [LaToza 2010]. From the answers of 179 developers at Microsoft, the authors collected 371 questions like “Are the benefits of this refactoring worth the time investment?”, “Is this functionality already implemented?” or “How does this code interact with libraries?”. The authors classified these questions in 10 categories: rationale, debugging, policies, history, implications, implementing, refactorings, teammates, building and branching, and testing. They concluded that having a better understanding of developers' information needs may lead to new tools, programming languages, and processes that make *hard-to-answer* questions less time consuming or error prone to answer.

Study on Pull-Request. Gousios *et al.* performed a study focused on the quality model that developers have in mind when they accept pull-requests on GITHUB [Gousios 2014]. Some questions characterize the projects (frequency of pull-requests, tools used to assess and perform the merge, kind of requests). Then they asked developers to rank factors of acceptance or rejection: presence of tests, number of commits, comments, etc. They asked how the code is reviewed, how the pull-requests are sorted. The work style of the developer is also considered. While the poll focuses on pull-requests, it is difficult to classify the underlying questions according to different perspectives.

Study on Integration Decisions. Phillips *et al.* performed a study focused on how developers of a large-scale system make branching and integration decisions while managing releases [Phillips 2012]. They evaluated a survey they previously elaborated [Phillips 2011] by conducting semi-structured interviews with seven developers of a company. The authors found that developers making decisions need to consider 10 factors, such as potential conflicts, bug counts, and dependencies between branches. The authors also identified the information needed to support integration. Release decision makers need to predict storms of conflicts, detect pressure building up from non-integrated changes, monitor code flow between branches (what is the frequency of integrations), and track branch health (metrics such as test results, bugs, and task completion at branch level).

Empirical Study on Branching and Merging. Premraj *et al.* presented an empirical study that observed developers branching without considering the consequences on merging [Premraj 2011]. The goal of the study was to understand the implications of such branching for the cost of merging changes. The study had two parts: 1) A qualitative study where 16 developers were surveyed (5 questions oriented to branchers and 3 questions oriented to integrators) to learn their views on branching and merging files, and their experience with the development overhead from branching and merging; 2) a quantitative study that calculated the number of branches, the number of merges on a number of files, and the time spent on merging files. From the study they established (a) the roles of the branchers and mergers (*i.e.*, architects, configuration managers, integrators and developers), and (b) the types of files that dictate the cost of merging (*e.g.*, configuration files). They concluded that VCS tools and VCS best practices (*e.g.*, *branch only when necessary, branch late, propagate early and often*) are not sufficient to share files in an agile development environment. They also suggested that contents of shared files must be aligned with the responsibilities of the primary owners of those files, as a way to decrease conflicts of branching and merging files.

Questions related to Evolution Tasks. Sillito *et al.* proposed a catalog of 44 types of questions programmers ask during software evolution tasks [Sillito 2008]. The authors aim to understand what a programmer needs to know about a code base when performing a change task, how a programmer goes about finding that information, and how well today's programming tools support evolution. They performed two qualitative studies [Sillito 2005, Sillito 2006] observing 9 and 16 programmers respectively, making changes to medium and large codebase. From the analysis of the empirical information collected during both studies, they established the used tools, types of change tasks, paired versus individual programming, and the level of prior knowledge of the code base. 44 questions were classified in 4 categories: (a) finding focus points (*e.g.*, "Where in the code is the text in this error message or UI element?"), (b) expanding finding points (*e.g.*, "Where is this method called or type referenced?"), (c) understanding a subgraph (*e.g.*, "How are instances of these types created and assembled?"), and (d) questions over groups of subgraphs (*e.g.*, "What will the total impact of this change be?"). They also established that 34% of the questions was fully addressed by tools and 66% of the questions only partially addressed. From the results, they found that programmers need better tool support for asking more refined or precise questions, maintaining context, and piecing information together.

Information Fragment Model. Fritz and Murphy conducted a study in which they interviewed 11 professional developers to identify different kinds of questions they need answered during development, but for which support is

weak [Fritz 2010]. From the results, they established a catalog of 78 questions classified in several categories such as: (a) people specific (12 questions *e.g.*, “Which code reviews have been assigned to which person?”); (b) code change specific (35 questions *e.g.*, “What are the changes on newly resolved work items related to me?”); and (c) work item progress (11 questions *e.g.*, “Which features and functions have been changing?”). Alongside this study, they introduced the information fragment model (*i.e.*, a subset of development information for the system of interest) and associated prototype tool built on top of ECLIPSE for answering the identified questions by composing different kinds of information needed. This model provides a representation that correlates various software artefacts (source code, work items, team membership, comments, bug reports, and others). By browsing the model, developers can find answers to particular development questions.

Catalog of Integrator’s Questions. Uquillas-Gomez presented a catalog of integrator’s questions in her Ph.D. dissertation [Uquillas Gómez 2012a]. To compile a list of questions, the author conducted an open call to the developers of three SMALLTALK communities to compile the questions (VisualWorks Users¹, European Smalltalk User Group², and PHARO project³). In such call, she requested integrators in the communities what questions they ask themselves when integrating changes. 20 integrators responded the call in a period of 10 days. The participants integrated changes on small, medium and large SMALLTALK projects. Moreover, Uquillas-Gomez took into account related studies presented above [Fritz 2010, LaToza 2010, Sillito 2008], extending the findings from the call with 8 extra questions taken from these studies. Finally, a PHARO integrator helped her to refine the questions, yielding to a catalog of 64 questions.

As summarized in this section, several related works present catalogs of questions as a means to understand development activities (*e.g.*, maintenance or code comprehension) and to identify the developers’ information needs. However, these works: (1) except Uquillas-Gomez’s work, in general they do not focus on integration activities specifically but on development activities in general; (2) they do not provide a qualitative analysis to understand what are the key integration activities without comprehensive tool support.

¹vwnc@cs.uiuc.edu

²esug-list@lists.esug.org

³pharo-project@lists.gforge.inria.fr

2.2 Methodology

Integration of changes is a difficult and tedious activity. We want to tackle real-world integration problems, thus we want to know the key (most relevant) integration activities with little tool support. We found that there is no qualitative nor quantitative information in that regard in literature. Thus, we established a 2-steps investigation to learn about integration activities: As a first step, we prepared a catalog of questions that integrators ask when performing integration of changes. By knowing which real questions are raised during integration and are troublesome to answer, we can identify what activities need more tool support. As a second step, we performed a study to discover which are the key integration activities and the level of tool support for each.

First Step: Prepare a Catalog of Integrator's Questions

The elaboration of the catalog for validation is result of a collaboration with Verónica Uquillas-Gomez, and based on a catalog published in her Ph.D. dissertation [Uquillas Gómez 2012a] (described in Related Work). The original catalog was too long for a survey, therefore we carefully condensed some redundant questions. The result was a catalog composed by 46 questions.

Second Step: Rank Integrator's Questions

The questions of the elaborated catalog represent integrator's needs to perform their everyday's activities. Such integration questions may have different importance, and some of such questions may be currently supported by development tools whereas others may not. Then, we conducted a survey to quantitatively rank each question of the catalog in two dimensions:

- *Importance*: Nothing, Little, Moderate, and Extreme.
- *Tool support*: No, Partially, and Yes.

We called for participation in several software development communities, which include SMALLTALK-related mailing-lists, the Twitter accounts of the Apache Software Foundation and the Eclipse Foundation. In a period of 5 months we received the responses of 42 participants who integrate changes on very diverse software projects. The survey included a "Participant Profile" part to categorize participants and their projects. We start by structuring the results of this part because of its impact on the results. The full and detailed results of this study are available in a technical report [Dias 2014].

2.3 Catalog of Integrator's Questions

Following, we present our catalog of integrator's questions. The questions are grouped into 5 categories: (a) authorship/ownership, (b) structural change characterization, (c) behavioral change characterization, (d) bug tracking infrastructure, and (e) temporal and change sequence. We briefly describe each category prior to introducing its respective questions. Each question is accompanied by an identifier (e.g., A_1) that is used to refer to the question in later sections.

Authorship/Ownership. The first category of questions is related to the owner of the original code, to the author of the changes, and to the committer. These questions assess the author's quality and the reliability level of his changes.

Authorship/Ownership questions

- A_1 Who is the author of this changed code?
 - A_2 Who was the previous owner of the changed code?
 - A_3 Has my own code been changed?
 - A_4 What is the general quality of the change committer?
 - A_5 How many people have contributed to this group of commits?
-

Structural change characterization. The second category of questions is related to the structure of the original code as well as the changes. They cover various aspects in terms of volume, impact volume, dependencies (which packages, classes should be loaded before), and so on. From that perspective, they are not tailored to a sequence of changes but more to a single commit [Uquillas Gómez 2010b].

Structural change characterization questions

- S_1 How large is the change?
 - S_2 How many entities (packages/classes/methods) are impacted by the commit? (Impacted in the sense they can stop compiling, for example)
 - S_3 Is this commit confined to a single package or spread over the entire system?
 - S_4 What is the complexity of the changes?
 - S_5 Do all the changes within the commit belong together? (Can we split the commit?)
 - S_6 Are there other packages that will need to change as well to integrate this commit? (Can we identify the users of the changed code?)
 - S_7 Will the code compile after applying this commit?
 - S_8 Is the commit conflict free? (Does this change generate any syntactic merge conflicts when integrating?)
-

continued on next page...

...continued from previous page

Structural change characterization questions

- S*₉ Which entities (packages/classes/methods) have been changed?
*S*₁₀ Does this change depend on other changes (in the source branch) to be functional (in the target branch)?
-

Behavioral change characterization. The third category of questions is related to the nature, behavior and intent of a change. Such questions can be mostly applied to changes within a single delta. Note that some of these questions are open-ended and therefore inherently difficult to answer automatically. Moreover they may require up-front knowledge of the system as well.

Behavioral change characterization questions

- B*₁ Does this commit follow rules and conventions?
*B*₂ Is the vocabulary used in the commit consistent with the one on the system?
*B*₃ Does this commit improve the quality of the system?
*B*₄ Does this commit correctly fulfil its goal? (Does it fix correctly a particular problem?)
*B*₅ What is the intention of this commit?
*B*₆ In a commit with 'strange code', was the strange code intentional (it has to be like that to turn around a special aspect of the system), or accidental (the author did not really know what he was doing)?
*B*₇ What kind of commit is it? (Bug fix/New feature/Refactoring/Documentation/...)
*B*₈ Does this commit fix/break tests? Which tests?
*B*₉ Is the commit covered by tests? What is the coverage? How can I test it?
*B*₁₀ If I apply the commit, what are the parts of my current system that it affects? What are the users (classes/methods/functions) potentially impacted by this change in the destination branch/fork?
*B*₁₁ What are the implications of this commit on the (potentially undeclared) API? (Are there any unknown users of the API that will be impacted by the changes?)
-

Bug tracking infrastructure. The fourth category of questions is related to bug tracking traceability of changes.

Bug tracking infrastructure questions

- I_1 To which bug entry does this change relate?
- I_2 What bug fixes also affected the part of the system that is being impacted by this change?
-

Temporal and change sequence. The final category of questions is related to situating changes within the context of a sequence of changes, as well as to the time at which the changes occurred. Indeed, often a change does not happen in isolation, other changes may depend on it and fork analysis requires to understand change dependencies [Uquillas Gómez 2014]. In particular, when working on a sequence of changes, these questions capture the place of a change within the sequence.

Temporal and change sequence questions

- T_1 How old is this commit (compared to the version to which it should be integrated)?
- T_2 In which commit/version of the system was this method/function previously changed?
- T_3 Did this class/method/function change (a lot) recently/in the past?
- T_4 Is this change to a class/method/function the most recent one (in the branch)?
- T_5 Is there any pending change in the sequence of commits (in the branch) that supersedes this one?
- T_6 Is this commit part of a whole series of commits?
- T_7 Does this commit depend on previous ones? (What are the other commits needed first to merge this commit?)
- T_8 Is the change to a class/method/function ever used in subsequent changes?
- T_9 Is this change to a class/method/function reverting the code to an old state?
- T_{10} What else changed when this code was introduced or modified (*e.g.*, documentation, website, database schema)?
- T_{11} What other classes/methods/functions changed when this code was introduced or modified?
- T_{12} What are the changes made by the same authors/during the same time period?
- T_{13} Did the changing classes/methods/functions of this commit change together in a previous commit?
-

continued on next page...

...continued from previous page

Temporal and change sequence questions	
T_{14}	If there were changes to class/methods/functions happening together in the past, can we suspect that there is still something missing in the current commit?
T_{15}	Were the classes/methods/functions affected by this change renamed in the past and, if so, in which version of the system?
T_{16}	What were the users (callers) of a changed method/function in a particular version of the system?
T_{17}	What are the current users (callers) of a changed method/function?
T_{18}	What commits of another branch have been integrated into this branch?

2.4 Results

In this section we present a summary of the results from our survey. We start by an analysis of the integrator's profiles and systems they work on. Next, we analyze the ranking of integrator's questions from the catalog. We remind that the detailed report of the collected data is available as a technical report [Dias 2014]. Finally, we analyze threats to validity of our results.

2.4.1 Participant and System Profiles

The results show that participants' experience is quite diverse and serious, both in development (Figure 2.1) and in integration (Figure 2.2). Compared with development experience, integration experience has a smaller spread period: from 2 to 20 years compared to the 5 to 45 years.

Most of the participants (88%), answered they are (or they were) developers of the system where they integrate changes; this is a result that we expected due to the complexity of integration activity, which requires a deep knowledge of the codebase of a project that generally only a developer working on it has. Turnover and open-source projects may change such fact.

System characterization. Most of participant's systems involve between 3 and 16 developers (Figure 2.3). About the frequency of integrations, participants answered performing them mostly *on demand*, *i.e.*, not regularly but when necessary. Around 20% of the participants answered that they never perform integrations between forks. In the answers, approximately half of the systems are open-source. About the size in lines of code (LOC), most of the projects have between 10k and 450k LOC. When we asked for the kind of software where developers integrate

changes, three quarters answered they work in *End user applications*, and the same number of responses for *Libraries, frames and platforms*. This means that many participants integrated changes both in client- and in provider-side of their software.

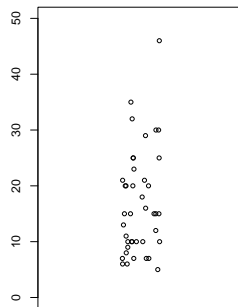


Figure 2.1: How many years have you been *developing* software?

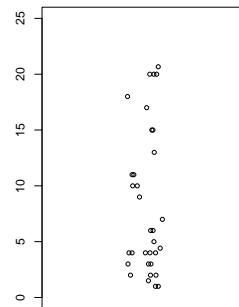


Figure 2.2: How many years have you been *integrating* changes?

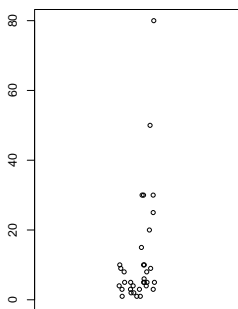


Figure 2.3: How many *de-*
velopers are working on this system? (we removed two outliers: 250 and 600)

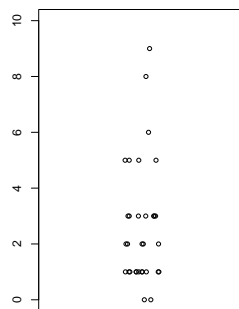


Figure 2.4: How many *inte-*
grators are working on this system? (we removed one outlier: 60)

Approximately 26% of participants answered they are not the main integrator of their system. This percentage shows how much the role of integration is shared by several project members. In fact, Figure 2.4 reveals that there is a median of 2 integrators per system. 93% of participants reported that they interact with developers when integrating changes. In the survey we asked what are the reasons of such interaction with developers. The main reason participants answered was solving merging conflicts. They answered as well that understanding the changes, and giving feedback about the quality of the changes are other reasons for interacting. Figure 2.5 shows that “Development” and “Release” are the most used types of branches, although “Feature”, “Bug fix” and “Experimental/Prototype”

are common as well. Figure 2.6 shows that the participants consider merge conflicts and regressions as the most significant problems. In the results, we observe the same number of participants use general VCS (CVS, Git, SVN and TFS) and SMALLTALK-specific VCS (MONTICELLO, StORE and ENVY) (Figure 2.7).

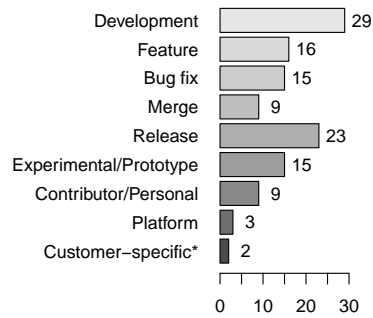


Figure 2.5: What types of branches are defined for this system? The "*" category (*i.e.*, "Customer specific") was extracted from "Other" field.



Figure 2.6: What are the most significant problems that you have had with merges?

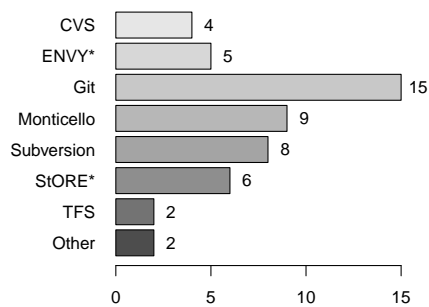


Figure 2.7: Which versioning tool(s) are using for this system? The "*" categories (*i.e.*, "StORE*" and "ENVY*") were extracted from "Other" field.

2.4.2 Integrator's Questions

In this section, we classify the questions and report only the questions identified as important and with little tool support. From this analysis we will get insights on how to improve the integration toolset.

Importance: If a question's responses are concentrated among No and Little importance, we say the question has *Agreed No-Importance*; if responses are concentrated between Moderate and Extreme importance, we say the question has *Agreed Importance*. Instead, when there is not a clear agreement among the answers, we say the question has *Disagreed Importance*.

Tool Support: If a question's responses are concentrated around No, we say the question has *Agreed No-Support*; if responses are concentrated around Yes, we say the question has *Agreed Support*. Instead, when there is not a clear agreement among the answers, we say the question has *Disagreed Support*.

We illustrate our classification criteria in Figure 2.8, which presents the responses to two integration questions:

A_1 : Who is the author of this changed code?

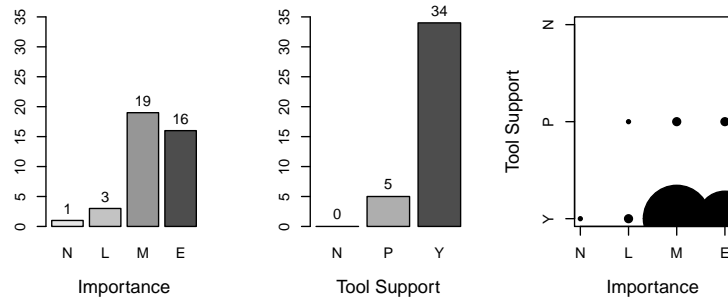
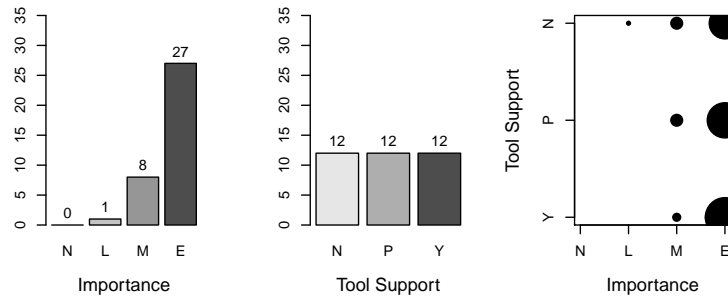
B_8 : Does this commit fix/break tests? Which tests?

For A_1 , participants mostly agreed that this question has from moderate to extreme importance, and that it does have tool support. Thus, we classify A_1 as a question with *Agreed Importance* and *Agreed Support*. For B_8 , participants also agreed that it is an important question, but they disagreed in the tool support. Then, we classify B_8 as a question with *Agreed Importance* and *Disagreed Support*.

We applied these criteria to all the questions of the catalog. Results are presented in Table 2.1. Overall, in the dimension of Importance, 33 questions (72%) have *Agreed Importance*, 10 have *Disagreed Importance* (21%), and only 3 have *Agreed No-Importance* (7%). In the dimension of Tool Support, 12 questions (26%) have *Agreed Support*, 9 have *Disagreed Support* (20%), and 25 have *Agreed No-Support* (54%). In the following section, we use this categorization to discuss the most relevant results.

2.4.3 Threats to Validity

The result of the first step of our research work, *i.e.*, the catalog of integrator questions, was originally compiled from surveying SMALLTALK community members, and thus it might be biased to the integration of changes in SMALLTALK projects. However, integration tools used in these SMALLTALK communities have similar characteristics than in other communities. Then, this fact should not represent a threat for our catalog.

(a) A_1 Who is the author of this changed code?(b) B_8 Did this commit fix/break tests? Which tests?

Title	Legend
Importance	Nothing (N), Little (L), Moderate (M), Extreme (E)
Tool Support	No (N), Partially (P), Yes (Y)

Figure 2.8: Examples of our classification criteria for two participant's responses.

The results of our second research step, *i.e.*, the survey responses, indicate that the participants profile were diverse in a number of relevant aspects (Section 2.4.1): development and integration years of experience, used VCS tools. Also, the participants work on both academic and industrial SMALLTALK projects that follow different development policies. In addition, the profiles showed that participants's experience includes several programming languages. Finally, we think the population size of this survey (42 integrators) is enough to extract reliable general conclusions even though we let as future work to repeat the survey with a larger population.

	<i>Agreed No-Importance</i>	<i>Disagreed Importance</i>	<i>Agreed Importance</i>
<i>Agreed No-Support</i>	T_{12}, T_{13}	$A_4, B_2, B_6, T_{14}, T_{15}$	$S_4, S_5, S_6, S_{10}, B_1, B_3, B_4, B_9, B_{10}, B_{11}, I_2, T_5, T_6, T_7, T_8, T_9, T_{10}, T_{16}$
<i>Disagreed Support</i>	A_5		$S_2, S_7, B_8, I_1, T_3, T_{11}, T_{17}, T_{18}$
<i>Agreed Support</i>		A_2, A_3, S_1, T_1, T_2	$A_1, S_3, S_8, S_9, B_5, B_7, T_4$

Table 2.1: Classification of the questions according to surveyed integrators.

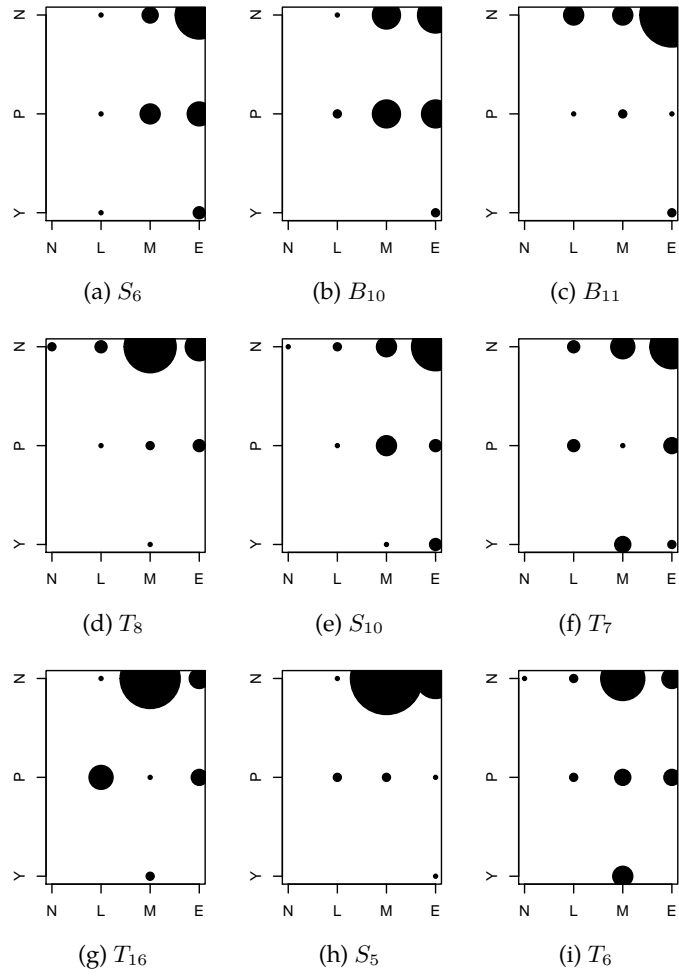
2.5 Discussion: Top Important Questions without Tool Support

The goal of the survey is to identify key integration activities without comprehensive tool support. To that end, we performed a thorough interpretation of the 18 questions identified as important with little tool support (in the upper-right corner of Table 2.1). The result was a selection of 9 questions (summarized in Figure 2.9). We found three conceptual axis in these questions, which we use to interpret the survey results: *understanding change impact*; *understanding change dependencies with cherry-picking*; and *understanding change scattering*.

Understanding Change Impact

Understanding the impact of applying a code change on the current system is a key concern of integrators. The effects of a change are of crucial importance since it can introduce unexpected behavior in the system. The following questions capture the problems faced by integrators when assessing the impact of code changes.

- S_6 : Are there other packages that will need to change as well to integrate this commit? (Can we identify the users of the changed code?)
- B_{10} : If I apply the commit, what are the parts of my current system that it affects? What are the users (classes/methods/functions) potentially impacted by this change in the destination branch/fork?)
- B_{11} : What are the implications of this commit on the (potentially undeclared) API? (Are there any unknown users of the API that will be impacted by the changes?)



Axis	Title	Legend
x	Importance	Nothing (N), Little (L), Moderate (M), Extreme (E)
y	Tool Support	No (N), Partially (P), Yes (Y)

Figure 2.9: Top 9 questions with *Agreed Importance* and *Agreed No-Support*.

T_8 : Is the change to a class/method/function ever used in subsequent changes?

T_{16} : What were the users (callers) of a changed method/function in a particular version of the system?

These five questions share the need of understanding the impact of integrating a change in the destination branch. However, these questions do not refer to the same kind of impact: In one hand, B_{10} and T_8 refer to the local impact, *i.e.*, which are the entities in the codebase (in the target branch) that are affected by the change. In the other hand, S_6 and B_{11} refer to the external impact, *i.e.*, which entities in other codebases are affected with the change. The latter is usually called *ripple effect*. [Yau 1978]

Questions T_8 and T_{16} complement each other: the former talks about impact in future commits of a branch, while the latter talks of impact in past commits. These questions address important issues when the commits around a change have to be understood.

Understanding Change Dependencies when Cherrypicking

Sometimes an integrator has to apply in a branch (*i.e.*, target or destination branch) some code changes selected from another branch (*i.e.*, source branch). This action is known as *cherrypicking* code changes. The main difference with a plain merge is that not every change from the source branch is applied in the destination branch, but only a selection of such changes.

Cherrypicking is a difficult activity: consider cherrypicking a modification in a method which adds a reference to a class that does not exist in the destination branch (leading to compilation errors). In general, a change can depend on other changes in the branch, and the identification of such dependencies is difficult. The following questions capture such problems.

T_7 : Does this commit depend on previous ones? (What are the other commits needed first to merge this commit?)

S_{10} : Does this change depend on other changes (in the source branch) to be functional (in the target branch)?

These two questions have much in common, since both focus in the understanding of the dependencies of a change. The main difference is in the granularity of changes: while T_7 is clearly a commit granularity, S_{10} dissolves the commit boundaries when it references to changes in general.

Understanding Change Scattering

Developers often bundle changes of unrelated tasks (*e.g.*, bug fix and refactoring) in a single commit, thus creating a so-called *tangled commit*. In a study, Herzig and

Zeller [Herzig 2013] analyzed several open-source projects and found that 20% of the bug-fixing commits are tangled, *i.e.*, these commits contain unrelated changes apart of the bug fix changes. In other cases, however, developers perform a single task (*e.g.*, implementing a new feature) spread over several commits, which also poses difficulties to understand changes.

The following questions can be interpreted as two sides of the same problem: the wrong spread of code changes along commits.

S_5 : Do all the changes within the commit belong together? (Can we split the commit?)

T_6 : Is this commit part of a whole series of commits?

Question S_5 is related to understanding changes in a *tangled commit*. Question T_6 is about understanding changes related to one task, that are spread in several commits.

Question S_5 is the most important question without tool support according to the participants of our survey. Unlike the other questions discussed above, S_5 and T_6 have no direct reference to change dependencies or impact.

2.6 Conclusion

Since we orient our work to tackle real-world integration problems, we needed to know the key (most relevant) integration activities without comprehensive tool support. After an analysis of the literature, we found that there is no quantitative information at this respect. Thus, we performed an exploratory study to rank the importance of integration activities and the level of tool support for each.

The most important contribution of this chapter to our thesis results from the analysis of the collected survey responses, where we identified three key integration activities without comprehensive tool support:

1. *understanding change scattering;*
2. *understanding change dependencies when cherrypicking; and*
3. *understanding change impact.*

These results serve as guidelines to focus our efforts on new approaches to improve everyday's work of integrators.

3 FIRST-CLASS CODE CHANGES

Contents

3.1	Version Control Systems	26
3.2	Reconstructing First-class Evolution	27
3.3	Recording First-Class Evolution	28
3.4	Change Management in PHARO	29
3.5	EPICEA	31
3.6	EPICEA Model	31
3.7	EPICEA Tools	34
3.8	Conclusion	36

Introduction

Often software engineering researchers resort to the data available in VCS to uncover information about software systems. Examples include extracting migration rules [Hora 2013, Hora 2014], automatic bug repair [Martinez 2014], inference of association rules between software artifacts to prevent defects [Zimmermann 2005], discovering software quality issues [Eick 2001] and predicting which parts of a software system are fault-prone, to focus reviewing and testing efforts [Nagappan 2005].

Mining Software Repositories [Herzig 2010] is an empirical software engineering research field that exploits the information that developers produce, that is stored in software repositories (such as a VCS or an issue tracking system). By carefully using scientific methods, researchers extract laws and build approaches that end in better development tools [Zeller 2013].

However, empirical studies are only as reliable as the data they rely on. Most of these studies depend on the accuracy of the mined evolution data, which is threatened by noise and incomplete data [Herzig 2013, Robbes 2005, Negara 2012]. Following, we go deeper into the causes of these problems in VCS.

3.1 Version Control Systems

Most popular VCS, such as SVN, GIT¹ and MERCURIAL², hold the following properties:

State-based evolution. VCSs store little or nothing of the operations that lead from one version to the next one. They represent the evolution of a system as a graph of versions. Then, the operations from a version to another is computed by comparing the snapshotted files. However, some operations are not trivial to reconstruct, and can only be estimated.

Text-based evolution. VCSs version file and directory structures, and represent a file change as a set of inserted, deleted and updated text lines. Due to this property, generally the program entities of a codebase are represented as text files. Thus, a developer that wants to understand the evolution of a system has to mentally decode text-line operations that the VCS handle into program entity operations. The substantial semantic gap between the program entities (reality) and the files (representation) hinders understanding of a system's evolution [Uquillas Gómez 2014] and raises merging problems [Mens 2002].

Commit-based evolution. The commit time is the only moment in which the VCSs record actions of the developer to the codebase. However, an arbitrary amount of developer actions might take place before a commit: for example, what looks like a one-line change in a commit might have been the result of a bug-tracking session that lasted several days [Robbes 2005]. Negara *et al.* detected that 37% of code changes performed by the developer are *shadowed*, *i.e.*, overridden by subsequent changes in the same line, file and commit [Negara 2012]. Such shadowed changes can not be reconstructed from the VCS data. Then, VCSs loose important data from the system's evolution.

These properties of most popular VCS lead us to classify the related works of software evolution tools into two main groups of approaches:

- *Reconstructing first-class evolution* approaches, that tackle problems of *text-based evolution* by recovering evolution from VCS data as program entity and/or change models.
- *Recording first-class evolution* approaches, that start with a clean slate by recording fine-grained developer actions from the IDE to tackle problems related to *state-based evolution* and *commit-based evolution* properties.

¹Git: <http://git-scm.com>

²Mercurial: <http://mercurial.selenic.com>

Next, we describe such related works. We show that whereas in the approaches in first point mitigate the aforementioned significant semantic gap in VCS data *precise* evolution data, the approaches in the second point aim at obtaining a *complete* evolution data.

3.2 Reconstructing First-class Evolution

FAMIX change models. FAMIX [Ducasse 2000, Tichelaar 2000] provides first-class models for program entities of diverse languages such as SMALLTALK, JAVA, PYTHON and COBOL. First-class code change models have been proposed on top of FAMIX: HISMO and ORION.

Gîrba proposed HISMO [Gîrba 2005, Gîrba 2006], which uses FAMIX to model the history of a software system, providing facilities for reasoning over versions.

Laval presented ORION, a history model [Laval 2009, Laval 2011] that extends FAMIX to support larger systems and histories than HISMO. Its design allows a more optimal memory usage, by sharing program entity models between system versions. ORION provides an interactive tool for software reengineering that allows simulation of changes and compare their impact on the system.

RING change models. RING [Uquillas Gómez 2012b, Uquillas Gómez 2010a] is a first-class model for SMALLTALK program entities whose objective is to serve as a unified infrastructure for building IDE tools in PHARO. On top of RING, Uquillas Gómez *et al.* model history as first-class citizens in two complementary models: RINGH and RINGC [Uquillas Gómez 2012a]. Using these models, the author built tools for change and history analysis: TORCH and JET. TORCH [Uquillas Gómez 2014] is a visual tool that helps to understand changes based on semantic characteristics of changes, such as scope and size in terms of program entities involved (*e.g.*, package, class, method).

CHANGEDISTILLER and EVOLIZER, by Fluri and Gall *et al.*, process JAVA source code files to reconstruct code changes [Fluri 2007, Gall 2009]. They represent evolution as series of AST changes (insert, delete, move and update node). To extract changes between two versions of a codebase, their algorithm matches AST nodes between the versions and approximate what operations can transform one tree into the other. Falleri *et al.* proposed GUMTREE, which computes shorter edit scripts, and thus are closer to the original developer's intent [Falleri 2014]. All these approaches are inspired on an existing tree differencing algorithm [Chawathe 1996].

Several approaches recover refactorings from VCS histories and highlight the importance of this information to understand the evolution of a system.

Demeyer *et al.* infer refactorings by comparing two versions of a codebase. They propose an heuristic algorithm based on low-level software metrics such as method size, class size, and inheritance levels [Demeyer 2000].

Weissgerber and Diehl presented a technique to detect refactoring candidates with high recall and precision [Weissgerber 2006].

Dig *et al.* presented a technique based on program entity references such as method calls and type imports [Dig 2006].

Prete *et al.* presented REF-FINDER, a template-based approach that can infer a wide variety of refactorings [Prete 2010]. Their templates build upon semantic analysis performed on program entities, such as method calls and inherited fields.

3.3 Recording First-Class Evolution

Rather than mining evolution data from VCS repository, the following approaches instrument IDEs to access a more complete data. Instead of just analyzing the end result of several hours or days of work, tools can know what exact changes were performed, whether there were pauses in the development, etc., in order to perform a given task [Maalej 2014].

Lippe, in an early work, presented his approach which records operations from the IDE and implements an advanced merging algorithm [Lippe 1992].

CHEOPS [Ebraert 2007] records developer's actions using a change model built on top of the above-mentioned FAMIX model.

Robbes' SPYWARE captures developer fine-grained code changes in a centralized repository [Robbes 2008b]. SPYWARE records detailed changes such as a line added in a method, as well as more high-level changes such as some automatic refactorings. He proposes a multiple-language model and abstract syntax tree (AST) change operations, as well as some high-level operations like 'class rename'.

Dig *et al.* present MOLHADOREF, a VCS that is aware of program entities and refactoring operations [Dig 2008]. This tool mixes operation-based and state-based merging to capture the semantics of refactoring operations. It records the refactorings performed by the developers, and calculates deltas that represent other changes. Therefore, it can merge changes that involve a combination of logged refactorings and textual editing.

CoEXIST [Steinert 2012] preserves fine-grained intermediate states of the system during a development session, allowing back-in time browsing and execution. A distinguished feature of CoEXIST is the possibility of easily executing arbitrary

code in any past intermediate version of the system during a development session. CoEXIST is a step-forward from CHANGEBOXES [Zumkehr 2007], which is a similar approach but misses proper tools for navigating and manipulating the history (*e.g.*, search and replay changes). However, the evolution data that both approaches preserve is restricted to the IDE runtime since they miss an external persistence mechanism for the code changes. This limits the possibility of performing long-term history analysis.

In parallel, three similar approaches were developed which gather developer actions from ECLIPSE IDE:³ Negara *et al.* presented CODINGTRACKER [Negara 2012, Negara 2014], Ge *et al.* presented BENEFACTOR [Ge 2012] and Foster *et al.* presented WITCHDOCTOR [Foster 2012]. These tools monitor code changes and detect manual refactorings, *i.e.*, refactorings performed by the developer without using the IDE automated refactorings.

3.4 Change Management in PHARO

Since we decided to validate our thesis in PHARO community, we performed an analysis on the most relevant tools for change management in the PHARO IDE: MONTICELLO and CHANGE LIST.

MONTICELLO. In PHARO, as in other SMALLTALK systems, MONTICELLO is the most popular VCS. MONTICELLO is a distributed VCS, such as GIT or MERCURIAL, that enables a developer to version snapshots of the program, and easily branch and merge. To the effects of our work, MONTICELLO has no substantial differences with the other distributed VCS aforementioned:

- the intermediate states of the codebase in the development session are lost;
- developer's actions are not reified.

CHANGE LIST. Most SMALLTALK systems provide the CHANGE LIST tool, which acts as a tape recording of source code changes. This tool writes down to disk code changes immediately after any editing operation. For PHARO developers, CHANGE LIST is a useful complement to MONTICELLO. It allows navigation of the different versions of the program entities with finer granularity than a traditional VCS. Additionally, a CHANGE LIST can be replayed: if the developer forgets to save changes before quit, or the system execution is accidentally interrupted (*e.g.*, the virtual machine crashes or the process is killed), then the developer can explore the CHANGE LIST to replay unsaved changes.

³ECLIPSE: <http://www.eclipse.org/>

While `CHANGE LIST` has proven to be reliable over the years, it has the following problems:

State-based model. When a program entity changes, `CHANGE LIST` only records the new state of it. For example, an instance variable addition and class addition are indistinguishable for `CHANGE LIST`. This means that `CHANGE LIST` loses relevant information about software evolution.

Barely structured text format. The log is a text file where each new event is written at the end as an executable command. To recover the original event, the written command must be executed. The `CHANGE LIST` format was designed for flexibility for replaying changes, but not for managing changes as first-class change model. `CHANGE LIST` format hinders tools to use the recorded events, and a declarative format is more appropriate.

Flat. Some IDE events trigger code changes. For example, an automatic refactoring such as ‘extract method’ triggers additions and modifications of methods. As a result, for a tool it is impossible to determine if a code change was manually performed by the developer or if it was triggered by a high-level event. This leads to problems on an accurate recovery of the evolution of a program during the development session.

Discussion

In our thesis, we claim that integration tools need to handle the main concepts of software evolution as first-class citizens. Since relevant change information is lost by regular VCS tools, our model can not be recovered by mining commit repositories but necessarily by gathering interaction data from the IDE. Considering the aforementioned approaches, we define the following conceptual requirements for our approach, that we will use to validate our thesis using `PHARO` as a case study.

First-class code changes Our approach must represent as first-class citizens the `PHARO` code changes, in a language-specific manner, to have a narrow semantic gap between the `PHARO` code change as the programmer has in mind when working and the model representation. This will provide us more *precise* evolution data to validate our thesis. Additionally, our model should include other relevant developer operations in the IDE, like automatic refactorings or unit-test runs, because this will increase the precision of the model to represent the evolution of the codebase. Also, it should be possible to determine if a code change was manually performed by the developer or if it was triggered by a high-level event.

Record first-class evolution Our approach should gather evolution data from the IDE to build the first-class code changes, rather than reconstructing it from

VCS data. This will provide us more *complete* evolution data to validate our thesis.

Additionally, we have some other requirements that will be useful to validate our thesis in real-world development, in the PHARO community:

- Provide IDE tools to browse and manipulate the first-class code changes.
- Provide redo and undo operations. Starting from the same or similar system, the evolution data should be enough to reconstruct the state of the system at any point of the development session.

Following, we describe our first-class change model and associated tools that log software evolution.

3.5 EPICEA

In essence, EPICEA listens to developer actions taking place in the IDE and records them as instances of the fine-grained code change and IDE events model. EPICEA records complete information of these events (*e.g.*, whether a test run failed), including a timestamp. EPICEA records code change operations (add, modify, and delete classes and methods) every time the developer saves the code in the IDE tools. EPICEA is invisible to the developer when recording as there is no impact on performance. EPICEA stores the collected data as a sequence of serialized objects in text files. We developed EPICEA as an open-source project⁴ that can be an every-day tool in the PHARO toolset. The project started as a branch of NEWCHANGESYSTEM⁵ project and was deeply modified and extended afterwards.

3.6 EPICEA Model

Most object-oriented languages share main concepts and vocabulary, such as classes and methods. However, the presentation of this first-class model for changes deserves some words about the relevant details of PHARO program entities.

Category. In PHARO, the *categories* are containers whose goal is the organization of *classes* and *traits* in logic units. Categories have no semantic for the program execution.

Class. As in most object-oriented programming languages, in PHARO a class defines the behavior and state of its instances, by means of *methods* and *variables*. There are variables of several kinds: *instance*, *class* and *shared*.

⁴<http://smalltalkhub.com/#!/~MartinDias/Epicea>

⁵<http://smalltalkhub.com/#!/~EzequielLamonica/NewChangeSystem>

Trait. Traits support the reuse of method collections over several classes [Schärli 2003].

A trait is a reusable unit of behavior that can be composed with other traits using a small set of composition operations. Like classes, traits have *methods*; however they do not have variables. The methods of a trait become available for the classes (or traits) that declare to *use* such trait in their definitions.

Behavior. While classes and traits (generically called *behaviors*) represent different concepts in the language, they share many characteristics. First, behaviors are identified by their *name*, which is unique in the system. Also, behaviors have *methods*, *trait compositions*, and a *category*.

Protocol. Behaviors have *protocols* to organize their methods. A protocol is a tag used to logically group methods, *e.g.*, 'printing' or 'initialization'. They have no semantic meaning during execution.

Method. A method belongs to either the *instance-side* or the *meta-side* of a behavior. Methods are identified in the behavior by their *selector* (*i.e.*, name). While the *source code* is the main property of methods, they also belong to a *protocol*.

3.6.1 Code Change Model

EPICEA models relevant code changes (Figure 3.1) that can be performed in the PHARO IDE.

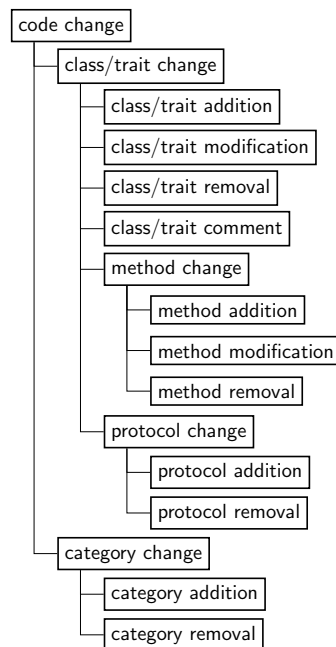


Figure 3.1: EPICEA code change model

Class/trait. The creation of a class or trait in the codebase (without methods) is a ‘class addition’ or ‘trait addition’, respectively, in EPICEA model. The ‘class modification’ represents every modification in the class definition, such as changing the superclass or removing a variable. A ‘trait modification’ is modification in the trait definition, like changing its category. The changes ‘class removal’ and ‘trait removal’ represent the deletion of a class or trait (with all its methods) from the codebase. A ‘class/trait comment’ models the change in the string of the comment of a class or trait.

Method. In EPICEA, the first compilation of a method into a behavior is a ‘method addition’. A change in either the source code or protocol of a method is a ‘method modification’. Finally, ‘method removal’ represents the deletion of a method from a class.

Protocol. The action of creating an empty protocol is modeled as ‘protocol addition’ in EPICEA. The inverse operation is a ‘protocol removal’.

Category. Similarly to protocols, the action of creating an empty category is modeled as ‘category addition’, while ‘category removal’ is the inverse operation.

PHARO provides an implementation of first-class program entities model, named RING (introduced in Section 3.1). The implementation of the EPICEA change model uses RING to represent the changing program entities, because *RING definitions* are useful to represent snapshots of a program entity. For example, a ‘method modification’ holds RING definitions of both the *old* and *new* versions of the changed method.

3.6.2 IDE Event Model

We claim that the evolution of a system can be better understood when tools have IDE interaction data from the development session. Therefore, we extend our first-class code change model with the following IDE events (Figure 3.2).

Refactorings. PHARO IDE provides a number of automatic refactorings such as renaming a method, a class, or a variable. The renaming of method *foo* to *bar* triggers several code changes: ‘method add(*bar*)’, ‘method removal(*foo*)’ and a ‘method modification’ for each method referencing the name *foo*. The ‘Refactoring run’ event models the execution of a refactoring.

MONTICELLO. EPICEA model includes the two most basic operations in the VCS tool: ‘Monticello save’ (commit a version to a repository) and ‘Monticello load’ (restore a version from a repository).

Tests. SUNIT, the xUnit testing framework⁶ in PHARO, is a widely-used tool. A ‘test run’ event models the execution of one or more tests in the IDE, and the

⁶xUNIT: <https://web.archive.org/web/20150315073817/http://www.xprogramming.com/testfram.htm>

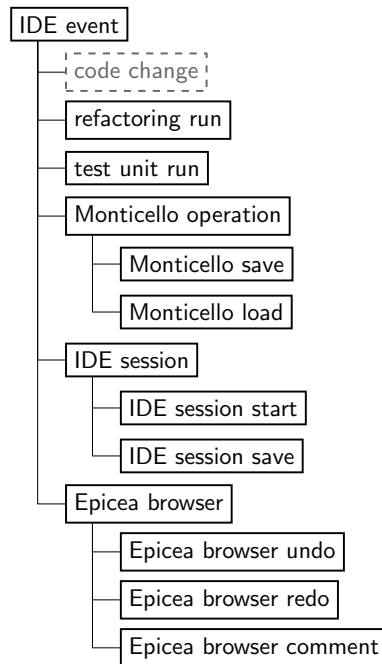


Figure 3.2: EPICEA IDE events model. We describe ‘code change’ in Figure 3.1.

results of each test: pass, failure, error, unexpected pass, expected failure, or expected error.

IDE Session. The PHARO IDE, as most SMALLTALK IDEs, works as a *snapshot*. A snapshot contains both the codebase but also the instances of the codebase under execution. When the developer *saves* the IDE (*i.e.*, ‘IDE session save’), the current state of the system is written down to disk. When the developer *opens* the IDE (*i.e.*, ‘IDE session start’), this means to restore a previously saved snapshot. In other words, the snapshot acts as a cache with preloaded codebase and initialized objects.

Epicea Browser. In Section 3.7 we will describe the EPICEA browser, an IDE tools where the developer can perform several operations that represented in the EPICEA model as with other IDE events: ‘Epicea browser undo’, ‘Epicea browser redo’, ‘Epicea browser comment’. For example, performing a ‘Epicea browser undo’ operation on a ‘method addition’ triggers a ‘method addition’.

3.7 EPICEA Tools

Our thesis has an empirical nature: we need to validate it with developers. In that sense, the developers from PHARO community are an important source of data and

feedback to evaluate our approach. Then, we provide a PHARO implementation that: (1) records EPICEA model events from the IDE; and (2) allows developers to access and use the recorded code changes and IDE events in EPICEA IDE tools.

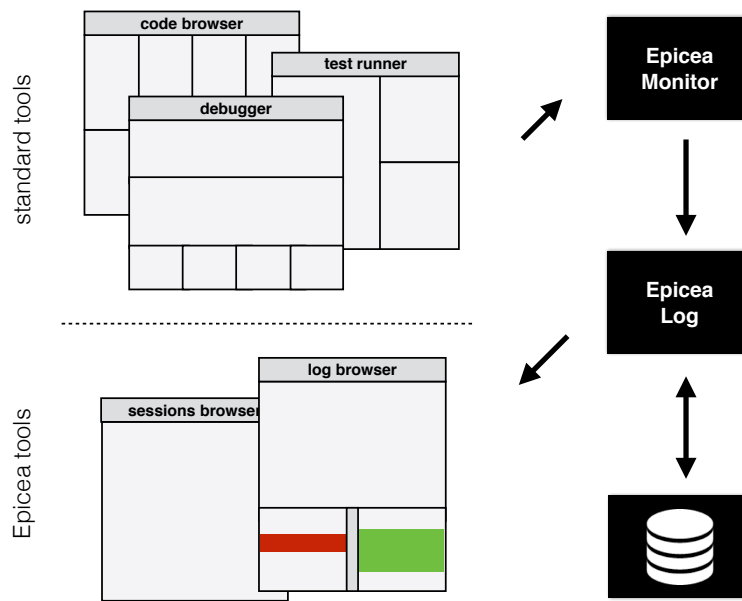


Figure 3.3: Overall architecture of EPICEA integration with PHARO.

EPICEA LOG. A log represents a collection of ‘IDE events’ and provides related operations such as registering new events and querying for past events. For each registered event, the log records contextual meta-information: the timestamp, the author, and potentially the event that triggered it (*e.g.*, undoing a method addition triggers a method removal).

EPICEA MONITOR. This is a pluggable extension to the PHARO IDE that listens to the actions of the developer to log them as the corresponding ‘IDE event’ instances. The monitor is subscribed to the *system announcements* provided by PHARO, and also instruments other IDE tools to have extra information (*e.g.*, the *test runner*).

The EPICEA implementation provides the following additional tools as extra features that enable EPICEA tools as part of the everyday’s work of a PHARO developer.

EPICEA LOG BROWSER. This browser is an IDE tool whose basic goal is to let the developer browse all the recorded events and filter them. Additionally, a developer can easily *export to* and *import from* files, which allows sharing portions of EPICEA logs with other developers. The developer can perform *undo* and *redo* operations using this browser, which might be useful during the development session. For example, the developer can rollback the codebase to a previous state by undoing last changes. Also, the developer can selectively undo or redo an arbitrary change. By means of the *comment* operation a developer can attach notes to EPICEA events.

EPICEA SESSIONS BROWSER. This browser provides the possibility to browse recorded events from log files. This tool is useful in the case of a developer that forgets to save changes before quit, because he can navigate and redo the lost changes.

3.8 Conclusion

Modern VCS tools loose relevant information of software evolution that, actually, can be gathered from IDE. We work on a new generation of integration tools that use such IDE information to better model code changes and build better integration tools. In this chapter we have presented our approach in this direction. First, we described the context in the practice of VCS, and analyzed related approaches in research, that included an analysis of the problems found in current PHARO system, where we want to validate our solution. Finally, we have presented EPICEA, a fine-grained code change and IDE events model and associated tools we will use in the next chapter to validate our thesis.

4 UNTANGLING CODE CHANGES

Contents

4.1	Problem Description	39
4.2	Proposed Solution: EPICEAUNTANGLER	42
4.3	Research Method	47
4.4	Results	53
4.5	Discussion	58
4.6	Related Works	60
4.7	Conclusion	62

Introduction

Version Control Systems allow programmers to control changes to source code and make it possible to find who made each software change, when, and where. This information is important to support both the coordination of developers working in teams [Guzzi 2015] and the creation of many recommendation and prediction systems related to software quality [Zimmermann 2005].

Developers often bundle unrelated changes (*e.g.*, bug fix and refactoring) in a single commit [Herzig 2011], thus creating a so-called *tangled commit*, such as the following taken from Jaxen:¹

```
-----
r1252 | elharo | 2006-11-09 [...] | 2 lines

Pulling getOperator up into BinaryExpr per Jaxen-169

[...]
Index: src/java/main/org/jaxen/expr/AdditiveExpr.java
=====
--- src/[...]/AdditiveExpr.java      (revision 1251)
+++ src/[...]/AdditiveExpr.java      (revision 1252)
@@ -61,7 +61,7 @@
 *
- */public interface AdditiveExpr extends BinaryExpr
+ */
+public interface AdditiveExpr extends BinaryExpr
 {
-     String getOperator();
 }

```

The tangled commit above contains both a refactoring (the move of `getOperator` to a different place, not shown in this extract), and code formatting (the move

¹<http://jaxen.codehaus.org, commit: svn-1252, 2006-11-09>

of an interface definition to its own line). Sharing tangled commits is problematic as they make code review, reversion, and integration harder and historical analyses of the project less reliable [Herzig 2013]. For example, even integrating the code formatting change included in the aforementioned commit (without the refactoring) would be a demanding task.

Untangling existing commits (*i.e.*, finding how to separate parts of a commit relating to different tasks) is an open research problem. Herzig and Zeller presented the earliest and most significant results in this area [Herzig 2011]: They implemented the first algorithm that can automatically untangle commits given artificially tangled ones.

In this chapter, we expand on this previous work by:

1. working in an untyped setting where a part of the approach by Herzig and Zeller is inapplicable;
2. considering fine-grained code change information gathered during development (*e.g.*, time at which each line has changed and all versions of each line); and
3. evaluating the resulting approach both on data generated by programmers who manually label it and with programmers working on real-world development tasks.

The ultimate goal of our work is to help developers of dynamically-typed code share untangled commits. To that end, we:

1. asked 7 developers to manually cluster changes for each of their commits using a dedicated tool, for a period of 4 months;
2. manually validated the generated data, selecting the data recorded by two of these developers, and computed a number of features based on their fine-grained code changes;
3. modeled the problem of predicting whether two fine-grained changes belong together, with a variety of machine learning approaches, determined the most appropriate one, and identified the most significant features;
4. designed an algorithm that uses the machine learning result to propose an automatic clustering of any tangled commit and developed a corresponding tool, `EPICEAUNTANGLER`; and
5. evaluated the effectiveness of our approach with developers who used `EPICEAUNTANGLER` in their daily work for two weeks.

Our results show that three features are especially important to perform clustering of fine-grained code changes: (1) the time between two changes; (2) the number of other changes between two changes; and (3) whether the two changes modify the same class. By modeling these features with Random

Forests [Breiman 2001], we identify whether two changes belong to the same commit with an accuracy of 95%, if training and testing on the same developer, and more than 88% if tested on a different developer. A set of 200 manually clustered fine-grained code changes (*i.e.*, the equivalent of a few days of work) was sufficient to reach good performance. When deploying EPICEAUNTANGLER with new developers during their daily tasks, we recorded an average success rate of 75% and a median one of 91%.

4.1 Problem Description

When developers want to share their work in a VCS, they will, more often than not, realize that they have done more than one activity, *e.g.*, fixed a bug, reformatted a method, and fixed a typo in a comment. Sharing everything in a single *tangled commit* is regarded as bad practice because it makes the following activities more difficult:

1. *Review* – Reviewers have to understand the code changes of all the activities *at once* [Tao 2012, Bacchelli 2013, Gousios 2014];
2. *Reversion* – Developers have to revert all changes of a problematic commit even when only the code change of one activity is problematic [Guzzi 2015];
3. *Integration* – Integrators have to merge or reject whole commits, *e.g.*, they will typically reject a code formatting operation and a bug fix included in the same commit [Uquillas Gómez 2012a];
4. *Historical analysis* – Researchers need to associate activities to files to conduct statistical analyses while, *e.g.*, mining software repositories [Herzig 2013].

4.1.1 Existing Solutions for Tangled Changes

To avoid tangled commits, developers could organize their work so that, at commit time, only one activity's code is to be shared. This requires frequent commits and interruptions in the developer's work flow [Beck 2000, Fowler 1999a, Steinert 2012]. Even with a lot of discipline, there will be times when a developer will have to split changed code into several commits.

To separate code from several activities into different commits, some tools (*e.g.*, `git add`) let the user select which files and lines to commit first. Being line based, these tools share the following problems:

1. The code present at commit time might be *incomplete* [Negara 2012]: Each change to a line shadows previous changes of the same line, thus making it impossible to commit the line as it was before the last change;

2. a commit resulting from a manual selection of a subset of all changed lines might be *invalid*: e.g., a developer might commit the beginning of a function definition but not the end; and
3. changed lines are shown in the order they appear in their files irrespective of their modification time: This makes it difficult for developers to select lines changed closely in time.

A great source of inspiration for us comes from Herzig and Zeller [Herzig 2011, Herzig 2013], who implemented an algorithm to automatically untangle commits. Their algorithm uses several *confidence voters* to decide whether two lines of a tangled commit should be put in the same cluster. They aggregate the results of each confidence voter into a single score, and then use the concepts of a multilevel graph-partitioning algorithm by Karypis and Kumar [Karypis 1995] to generate the clusters. Their voters include:

- **FileDistance**: the number of lines between the two lines if they are both in the same file;
- **PackageDistance**: the number of different package name segments within the package names of the changed files;
- **CallGraph**: the difference between the call graphs of the program with each line change applied separately;
- **ChangeCouplings**: the frequency with which the files both lines were changed into are committed together, using the work from Zimmermann *et al.* [Zimmermann 2005];
- **DataDependency**: a boolean indicating if the two lines read or write the same variable(s).

In the work by Herzig and Zeller we see the following limitations:

Dependence on static-analysis: The voters CallGraph and DataDependency rely on static analyses that might not be possible for dynamically-typed programming languages, or that might be available in a weaker form;

Incompleteness: The tangled commits used as input to the algorithm suffer from the incompleteness problem described earlier in this section: If a line is changed twice before a commit, the commit only contains the latest version of the line, shadowing a previous version of the line which could have been part of an untangled commit;

Artificiality: The validation by Herzig and Zeller relies on a classification of 7,000 existing commits done by the researchers without feedback from each project's experts. We believe that only the author of each commit can, at commit time, best organize his changes into untangled commits. Moreover,

the untangling algorithm by Herzig and Zeller relies on the knowledge of the expected number of untangled commits for a particular tangled one. With the goal of helping developers creating untangled commits, we do not have access to this information.

4.1.2 Addressing the Current Limitations

In our work, we propose to alleviate the aforementioned limitations by

- (a) expanding the setting to a dynamically-typed environment where some kinds of analyses are not available;
- (b) using fine-grained code changes that we collect during development sessions;
- (c) relying on developer-approved data for the validation of untangling approaches.

This results in the following requirements for the approach, EPICEA_{UNTANGLER}, that we present in this chapter:

The Dynamically-Typed Setting: Whereas the approach of Herzig and Zeller relies on static analysis of Java programs to untangle commits, our approach helps developers to create untangled commits in an environment that is dynamically-typed. Certain types of static analysis, *e.g.*, accurate call graph analysis, is not possible for dynamically-typed languages (JAVASCRIPT, RUBY, PYTHON, etc.). Therefore, our approach cannot rely on such static analysis.

Fine-Grained Changes: In modern integrated development environments (IDEs), tools can be notified each time a software artifact is changed and saved. As a result, a tool could listen to all fine-grained changes made by developers and, at commit time, present the developer a list of all the changes they have done. For example, a developer changing and saving the source code of a method 3 times will result in 3 fine-grained changes. This is in contrast with most tools that only present the latest version of each changed line; this requirement tackles the *incompleteness* limitation.

Developer-Approved Data: The untangling algorithm should be based on data created by developers who personally untangle the tangled commits that they produced in the first place. The final version of the approach should provide each developer, at commit time, with a list of the automatically untangled commits containing their fine-grained changes: Each developer could then reorganize these automatically-computed clusters of changes. Results must be validated by comparing the change clusters that are automatically computed against the reorganization done by the developer in a manual way.

4.2 Proposed Solution: EPICEA UNTANGLER

In a nutshell, our solution is to develop an approach and associated tools to help developers share untangled commits. The tools log all the fine-grained changes made by developers as they change the source code. When a developer wants to commit her changes, the tool, based on an analysis of the recorded information, presents several automatically-computed clusters of changes: Each cluster represents a distinct activity of the developer since last commit. The developer may then add a comment to each cluster and, if necessary, adapt the automatic clustering (by adding/removing clusters and moving changes to different clusters). Once the developer validates the clusters, the tool generates one commit per cluster and publishes them to the repository. In the following section, we present our solution decomposed in individual parts.

4.2.1 Gathering Fine-Grained Changes and IDE Events

Central to our approach is the collection of fine-grained information from the developer's IDE with EPICEA. Figure 4.1 summarizes the IDE events gathered by EPICEA used in our proposed solution. As in previous studies (both in ECLIPSE [Hattori 2010] and in SMALLTALK [Robbes 2007]) which showed that save-based recording produces reliable fine-grained code change data, EPICEA collects code change operations (add, modify, and delete classes and methods) every time the user saves the code in the IDE.

4.2.2 Voters

Once the data is collected, we have to characterize it in a way that it can be used for generating untangled changes. Similarly to Herzig and Zeller [Herzig 2011, Herzig 2013], as a first step we model our clustering task as a binary classification problem: For all the potential pairs of recorded fine-grained changes, we want to determine whether they belong in the same cluster. To this end, we implement a number of features or, maintaining the term used by Herzig and Zeller [Herzig 2011], *voters*, which describe different relations between the considered changes. Our voters (detailed in Table 4.1) span the following six dimensions:

1. *Code structure*: Although dynamic languages make it difficult to conduct static analysis, it is possible to compute basic relations. Our three voters in this dimension consider whether two changes happen in the same package, class, and/or method.
2. *Content*: This voter returns true if the two changes to a method are only source-code reformats, *i.e.*, if the *abstract syntax tree* of a method remains the

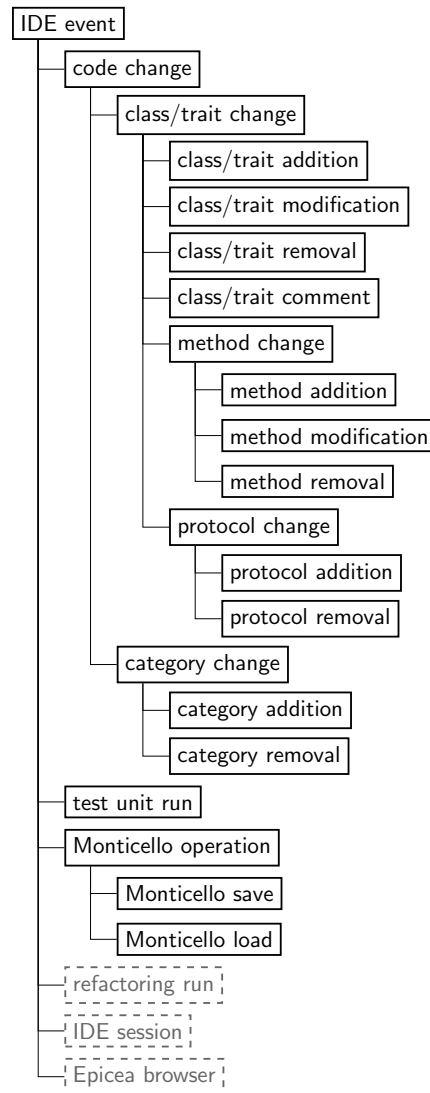


Figure 4.1: IDE events recorded by EPICEA. All events are considered by EPICEAUNTANGLER, except the grayed ones.

same after a change on it. This voter should help linking changes regarding refactoring actions.

3. *Testing*: EPICEA records test runs. The rationale of this voter is that two changes happening between runs of the same test could be related to the same task (*e.g.*, this should hold in the case of test-driven development).

4. *Spread*: These voters measure the distance between the two considered changes, considering time passed and number of other changes in between. We expect close changes to be more related.
5. *Message sending*: This dimension analyzes whether the changes involve related message sending (also known as ‘method invocations’, in languages such as Java or C#).
6. *Variable accessing*: This dimension computes relations between the variables accessed by the two changes: For example, a change that adds a new instance variable to a class may be related to a change that adds an usage of the same variable in a method.

The input of each voter is a pair of changes, and the output is of the type specified in column ‘Type’ of Table 4.1.

Voter Name	Dimension	Type	Relation between the two considered changes
samePackage	Code structure	Boolean	They involve the same package.
sameClass	Code structure	Boolean	They involve the same class.
sameSelector	Code structure	Boolean	They involve a method with the same name (regardless its class).
bothCosmeticChanges	Content	Boolean	They are both cosmetic (<i>i.e.</i> , pretty-printing—both versions of the method return the same result).
sameTestRun	Testing	Boolean	They are modified between the same unit-test runs.
numberOfEntriesDistance	Spread	Numeric	How close they are in the history, the voter computes number of other changes between them.
timeDifference	Spread	Numeric	How close in time they are in the history, the voter computes the seconds between them.
reciprocalMessageSends	Message sending	Nominal	They invoke each other; it computes 0, 1 or 2 if, respectively, no, one, or both call the other.
numberOfSharedMessageSends	Message sending	Numeric	They share a number of the same message sends.
numberOfSharedMessageSendsInDelta	Message sending	Numeric	They add or remove a number of the same message sends.
numberOfVariableAccesses	Variable accessing	Numeric	One change modifies or adds definitions of instance variables, the other accesses some of them.
numberOfSharedVariableAccesses	Variable accessing	Numeric	They access a number of the same instance variable names.
numberOfSharedVariableAccessesInDelta	Variable accessing	Numeric	They start or stop accessing a number of the same variable names.

Table 4.1: Different voters tested in our investigation.

4.2.3 Machine Learning Approaches

Our approach computes the values for each voter for each pair of changes (for performance reasons, we only consider fine-grained change pairs that are less than 3 days apart); to aggregate these values and train models that would predict whether two changes should be in the same cluster, we use machine learning (ML).

We consider three well-known machine learning algorithms that can handle binary classification [Hastie 2001]:

1. binary logistic regression (binlogreg),
2. naïve bayes (naivebayes), and
3. random forests [Breiman 2001] (ranforest).

We chose these algorithms not only because they have been applied successfully to a number of data mining tasks related to software engineering, but also because they make quite different assumptions on the underlying data and model (*e.g.*, naïvebayes relies on the conditional independence assumption, *i.e.*, the value of a voter is unrelated to the value of the others, and binlogreg requires each observation to be independent and linearity of independent variables and log odds), thus they can offer different interpretations. The choice of the most appropriate machine learning algorithm is based on the empirical data collected during the experiment.

This machine learning step takes as input the values computed by the voters for two particular changes, and it outputs the probability of the two changes belonging to the same cluster.

4.2.4 Clustering

The last necessary step in our approach is to take the output of the machine learning step, computed on each pair of changes, and aggregate it to form the clusters of changes for the user.

In this method, each change is initially considered to be a cluster of its own. Then pairs of clusters are successively selected by their maximum scores and merged. The result of this method is a *dendrogram*, which is a binary tree that represents the nested clustering of code changes. In this dendrogram, each non-leaf node has a *similarity level* that represents how similar are both children. In our problem, a similarity level of 1 corresponds to two clusters that must be merged, while a level of 0 corresponds to the opposite decision.

Finally, the desired clustering of code changes is obtained by cutting the dendrogram at some *similarity threshold*. Using a too low threshold produces too many small clusters, while a threshold that is too high produces a single cluster. The

choice of the most appropriate similarity threshold depends on the change set and, similarly to the machine learning approach, is based on the empirical data collected during the experiment.

The output of this step is the set of independent clusters of fine-grained changes, which is eventually displayed to the user with a dedicated user interface.

4.3 Research Method

In this section, we describe how we structure our research in terms of research questions, we present the research settings, and we outline our research method.

4.3.1 Research Questions

The ultimate goal of our work is to help developers of dynamically-typed code share untangled commits. For this we devise and test the approach we previously described to untangle code changes at a fine level of granularity. Accordingly, we structure our empirical investigation through the following three research questions:

RQ1: Which voters are significant to untangle fine-grained code changes?

With this question we aim to understand which are the most important voters in our untyped setting. To answer this research question, we consider the machine learning task of deciding whether two changes should belong to the same cluster. In doing so, we also determine which machine learning approach among the three we test, is better suited to model the problem through our voters.

RQ2: How effective is a machine learning model based on the significant voters in untangling historical fine-grained code changes?

Once we find the most significant voters and the best machine learning approach, we are interested to know their performance in predicting whether two changes should belong to the same cluster. We also want to investigate the effect asserted by individual developers' working styles on prediction performance; for this we train and test the machine learner on data generated by different developers (*e.g.*, training on one developer's data and testing on another developer's data).

RQ3: How effective is a tool based on the best voters and machine learning approach, when deployed with developers working on their daily tasks?

Finally, we want to devise an approach `EPICEAUNTANGLER`, based on the best machine learner and voters, to generate clusters and present them with a

graphical user interface. We want to test its effectiveness when deployed with participants

- (a) whose data should not have been used for training the classifier, and
- (b) who should be working on their usual development tasks.

4.3.2 Research Settings

The choice of PHARO as a case study was good for two main reasons:

First, the programming language, the development environment, and the versioning system are tightly integrated. This allows for a fast prototyping of an approach to record fine-grained code changes and interaction with testing and the versioning system.

Second, the PHARO open-source community of developers was been receptive to welcome and thoroughly evaluate research tools. This feature allow us to collect fine-grained data about code changes and IDE interactions from participants doing real-world development work. It also enabled us to deploy our resulting tool with more participants to evaluate its results.

4.3.3 Research Steps

4.3.3.1 Fine-grained data generation and collection

To answer our first two research questions, we need a *ground truth* to train and test our voters and machine learning approaches. Such a ground truth should be a reliable dataset containing fine-grained code changes correctly split into tasks by their authors. To obtain this, we contacted 7 participants actively contributing to PHARO, including the author of this Ph.D. dissertation. We asked them to install Epicea and to use the tool (*i.e.*, EPICEA TASK CLUSTERER (ETC), Figure 4.2) that we devised to manually cluster their fine-grained code changes. We showed a screencast² demoing the tool to all the participants before they started using it, so that they could understand the goal of the experiment and adapt their workflow accordingly. Every time the participants decided to commit their code to the versioning system, during their normal work, the ETC's interface would appear (as in Figure 4.2) with a list of all the fine-grained changes, since the previous commit, that the user had to manually cluster into tasks.

In detail, the main user interface of EPICEA TASK CLUSTERER (shown in Figure 4.2) works as follows: In the top pane, each column (*e.g.*, Point 1) represents a task (to group an activity of the user), and each item in a column represents a code change (*e.g.*, Point 2). Each code change is in a *ClassName»methodName* format, and the icon shows the type of change (as in Figure 4.1). The bottom pane

²Available at: <https://www.youtube.com/watch?v=fQVWuMQUBew>

Table 4.2: Participants' information

P_ID	current role	programming experience (in months)		
		overall	industrial	with PHARO
Data generation and collection phase				
P1	Ph.D. student	168	60	48
P2	Ph.D. student	48	36	24
Evaluation in real-world development phase				
P3	Ph.D. student	180	18	36
P4	software engineer	132	72	13
P5	associate professor	72	12	24
P6	Ph.D. student	72	11	11
P7	software engineer	180	10	30
P8	software engineer	60	18	36

(Point 3) shows the details of the selected change, in a *unified-diff* format. The user can review the listed changes and perform three actions to specify the expected clustering for them: Add a new empty task/cluster (Point 4), reopen an already closed task/cluster (Point 5), and move changes between columns (with drag and drop, Point 6). Once the clustering task is completed, the user presses the button Done, and the interface disappears.

4.3.3.2 Data analysis and evaluation of voters

Once the participants concluded the data collection period of 4 months, we conducted exploratory data analysis [O'Neil 2013] on the generated clustered changes. The data generated by five users was extremely sparse and inconsistent; these users confirmed this explaining that they could not afford the time required by EPICEA TASK CLUSTERER to review each change made during the experiment period. We removed this data and kept the data generated from the remaining two users (including the author of this Ph.D. dissertation) whose features are described in the top half of Table 4.2. Table 4.3 describes the resulting dataset (2devs).

Using 2devs we answered RQ1 and RQ2. As previously detailed (Section 4.2.3), we used machine learning to identify pairs of changes belonging to the same commit, by modeling it as a binary classification. For all potential pairs of changes in 2devs, we calculated values for all the voters in Table 4.1 and labeled with 'true' if

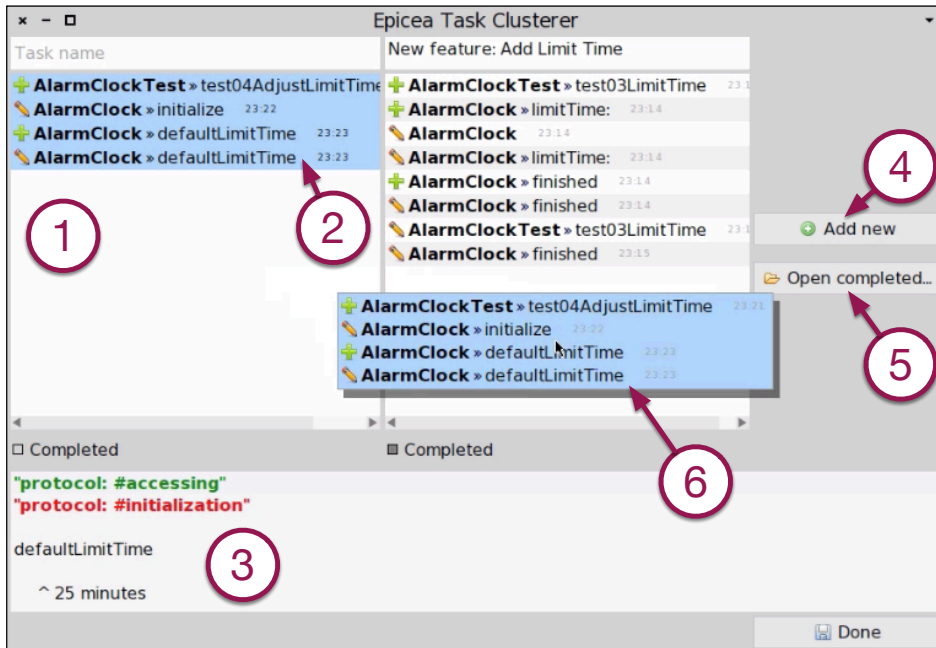


Figure 4.2: UI used in training stage. The user manually clusters the changes.

Table 4.3: Descriptive statistics of dataset 2devs

P_ID	Total number of		Changes per cluster			
	changes	clusters	Mean	Median	St. Dev.	Max
P1	15,175	298	50.9	8	153.1	1,582
P2	9,601	119	80.7	16	151.9	812

the changes belonged to the same commit or ‘false’ otherwise. As our dataset was unbalanced (the false class overruled the true one by a ratio of 4:1), we adjusted to avoid overfitting. Models were thus trained with a ratio of 2:1 samples for the false and true class respectively.

Evaluation of voters. To evaluate each trained model, we used standard machine learning metrics [Hastie 2001], such as precision (*prec*), recall (*rec*), accuracy (*acc*), the Area Under the receiver operating characteristic Curve (*auc*) and the F-measure (*f.measure*). Models were trained with an increasing number of samples as input (10^4 to 10^6 samples) to determine the minimum number of samples required to obtain adequate performance. At each input size, we used random selection 10-fold cross validation to evaluate model stability and reported results based on the mean of the 10 runs. We selected the best classifier and applied a

classifier-specific process to rank voters according to their importance in the classification. Then we incrementally trimmed the voter set starting from the least important feature until the performance of the classifier was severely impacted. Finally, we retrained the best classifier with the trimmed voter set and used that as our final prediction model. The final model was then exposed as a web service that EPICEAUNTANGLER used to drive the change unangling process to answer RQ3.

4.3.3.3 Deployment and evaluation with developers

Once we completed the creation and evaluation of the best ML approach and features on dataset 2devs, and obtained promising results, we created the corresponding implementation in EPICEAUNTANGLER, a tool that developers can use in real-world development.

During developer's work, EPICEAUNTANGLER records the fine-grained change information, exactly as done for the data collection phase. When the developer wants to commit, our approach computes the values for all the significant voters for each pair of code changes, and queries the web service implementing the final model of the ML classifier. For each pair, the web service returns a score between 0 and 1, indicating the probability that the two changes belong to the same cluster, according to the trained model. EPICEAUNTANGLER aggregates all the scores to form clusters using agglomerative hierarchical clustering method (see Section 4.2.4). This method outputs a dendrogram, which has to be cut at some similarity threshold to obtain the clusters of changes. We created a testbed with change set-expected clustering pairs whose purpose is to help us to conceive a good function for obtaining the similarity threshold for cutting the dendrogram. In Figure 4.3 we illustrate the function. The similarity threshold we chose corresponds to the maximum similarity gap between all nodes whose similarity level is less than 0.25. The intuition behind taking the maximum similarity gap is that continue merging code changes together is not worth, because the meaningful clusters have already been detected. The reason to use 0.25 as a lower bound is that we observed from data that such a low likelihood indicates in most cases changes that should not be merged.

This process happens in the background: After the developer decides to commit, she sees an interface similar to that used to generate the data for 2devs (Figure 4.2), with the difference that the clusters are already pre-computed by the tool. Then, the user browses the clusters and reorganizes the changes in case the pre-computed clusters are wrong.

Evaluation of clustering. To conduct this evaluation, we recruited six participants, whose features are described in the bottom half of Table 4.2. They all used EPICEAUNTANGLER for 2 weeks. To evaluate the clustering, each participant was asked to confirm whether the automatic clustering was correct; if not they could

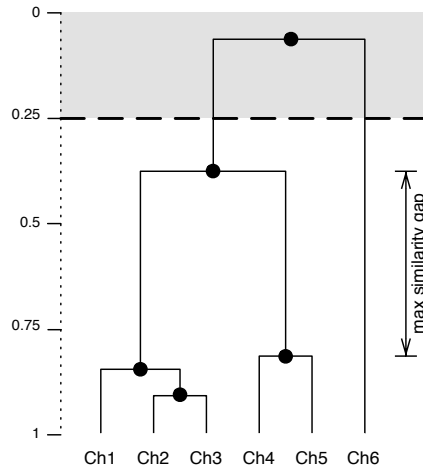


Figure 4.3: Determining the *similarity threshold* to cut the dendrogram.

rearrange changes to the correct clusters. We used the resulting data to evaluate the accuracy of our approach.

To measure the success rate of our approach, *i.e.*, how similar the *computed clustering* (from our algorithm) is to the *expected clustering* (from the developer), we need to know the ratio between the number of successfully clustered changes and the total number of changes. To know if a change has been successfully clustered, we must find which computed cluster best matches which expected cluster.

Figure 4.4 shows a sample comparison between a computed clustering and an expected clustering. The matrix on the right represents the *Jaccard indexes* computed for each pair of clusters; this index is defined as using the following formula:

$$J_{C_i E_j} = \frac{|C_i \cap E_j|}{|C_i \cup E_j|}$$

This Jaccard index represents how much two sets coincide. It ranges from 0 to 1, where 1 means the two sets are equal (*e.g.*, C_3 and E_1 in Figure 4.4) and 0 means the two sets have nothing in common (*e.g.*, C_4 and E_2 in Figure 4.4).

From the resulting matrix we want to know which computed cluster matches which expected cluster. This can be obtained by maximizing the sum of the Jaccard indexes over all permutations. For the sample in Figure 4.4 the maximum sum over all the permutations (3.5) is attained for this set of pairs:

$$Matching = \{(C_1, E_2)(C_2, E_4)(C_3, E_1)(C_4, E_3)(C_5, E_5)\}$$

We compute the success rate of our algorithm using the following formula:

	computed	expected		E1	E2	E3	E4	E5
C1	ch3	ch1 ch2	E1					
C2	ch5 ch6	ch3	E2	C1	0	1	0	0
C3	ch1 ch2	ch4	E3	C2	0	0	0	½ ½
C4	ch4	ch5	E4	C3	1	0	0	0
C5		ch6	E5	C4	0	0	1	0
				C5	0	0	0	0

Figure 4.4: Comparison between a computed clustering and an expected clustering. On the left-hand side, each box represents a cluster of changes. The computed clustering contains 4 clusters labeled from $C1$ to $C4$ (cluster $C5$ is a *virtual cluster* to ease comparison). The expected clustering has 5 clusters: $E1$ to $E5$. On the right-hand side, the matrix shows the corresponding Jaccard indexes.

$$SuccessRate = \frac{\#SuccessfullyClusteredChanges}{\#Changes}$$

A change ch_i is *successfully clustered* if the computed and expected clusters that contain ch_i are in the same pair of the *Matching* set. In Figure 4.4, all changes are successfully clustered except $ch6$. This gives us a success rate of $5/6 = 0.83$.

4.4 Results

In this section we answer our research questions, by describing the results we obtained in our evaluations.

4.4.1 What Are the Dominant and Significant Voters?

As a first step to answer our first research question, we use all the machine learning approaches we consider on the collected data and we evaluate whether an approach performs undoubtedly better. Table 4.4 reports the results of the classification performance of each machine learning approach for predicting whether two changes belong together, using a training size of $n = 320,000$ pairs (or 800 fine-grained changes), on the 2devs dataset. Overall, and across all metrics, the Random Forests algorithm delivers the best results, by a large margin. The high `REC` measurement of the binlogreg result can be justified by its equally low `PREC`; the classifier marks most of the file changes as belonging in the same cluster, but few of those decisions are correct.

Table 4.4: Classification performance on 2devs by approach

Classifier	AUC	ACC	PREC	REC	F.MEASURE	G.MEAN
binlogreg	0.92	0.68	0.43	0.96	0.60	0.76
naivebayes	0.88	0.65	0.41	0.94	0.57	0.73
ranforest	0.99	0.96	0.96	0.88	0.92	0.93
ranforest-trimmed	0.98	0.95	0.96	0.82	0.88	0.90

Once we established that ranforest delivers the best results, we assessed the importance of each voter for its classification result. We used the process suggested by Genuer *et al.* [Genuer 2010]. Specifically, we run the algorithm 50 times on a randomly selected sample of 10^6 change pairs, using a large number of generated trees (500) and trying 5 random variables per split. Then, we used the mean across 50 runs of the Mean Decrease in Accuracy metric, as reported by the R implementation of the *ranforest* algorithm, to evaluate the importance of each feature. The results can be seen in Figure 4.5. The three most important voters are:

1. the time difference between the changes,
2. the ordered distance of the changes,
3. and whether the changed code belonged in the same class.

We cannot make inferences about whether the effect of each voter is positive or negative to the response class; nevertheless, we believe that the results are indicative of the task-based nature of software development.

4.4.2 How Effective Is Random Forests with the Dominant Voters?

We answer our second research question by using only the three most important voters to train the prediction model. The prediction results are reported in Table 4.4, marked as ranforest-trimmed. We see that even with just those voters we obtain very good prediction results: The new model is within 3% of the performance of the model trained in all metrics.

Furthermore, we analyze the impact of the developer who made the changes on training and testing. We expect that behaviors of developers might be different and have a significant impact on the model.

We start showing that we obtain the best results when we train and test from data generated by the same developer (the intradev dataset in Figure 4.6). This confirms our hypothesis that the behavior of the specific developer has an impact on the model and the results. Furthermore, we see that the results are not equally

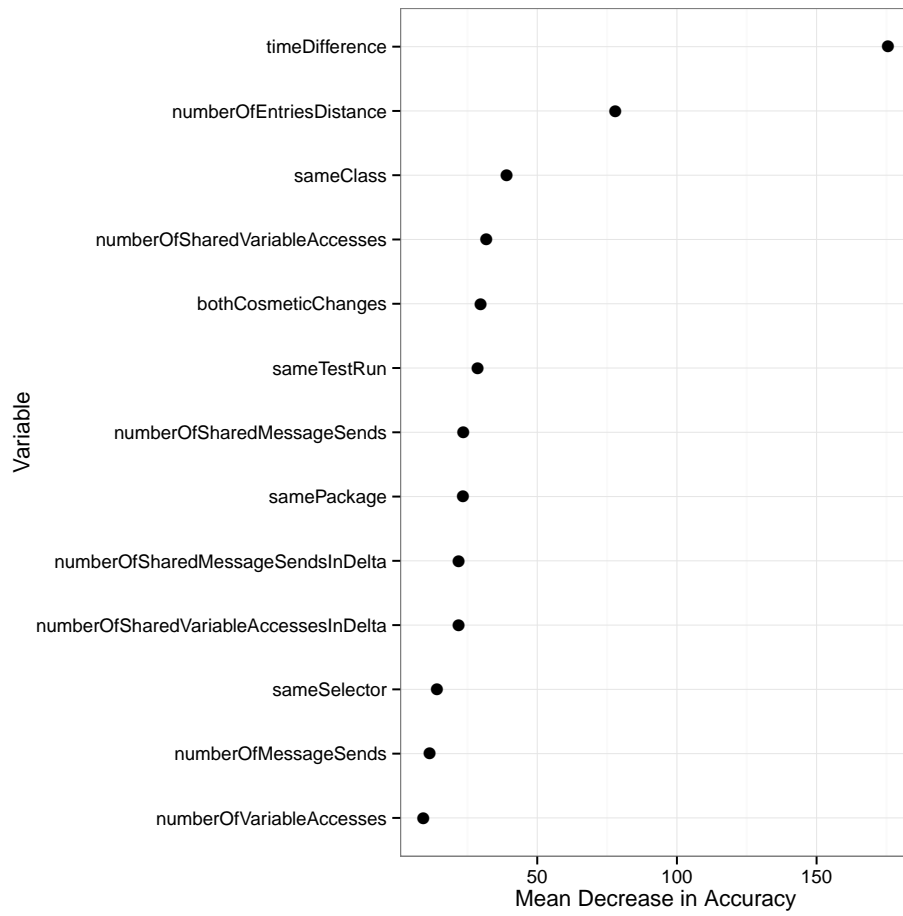


Figure 4.5: Voter importance for the random forest classifier.

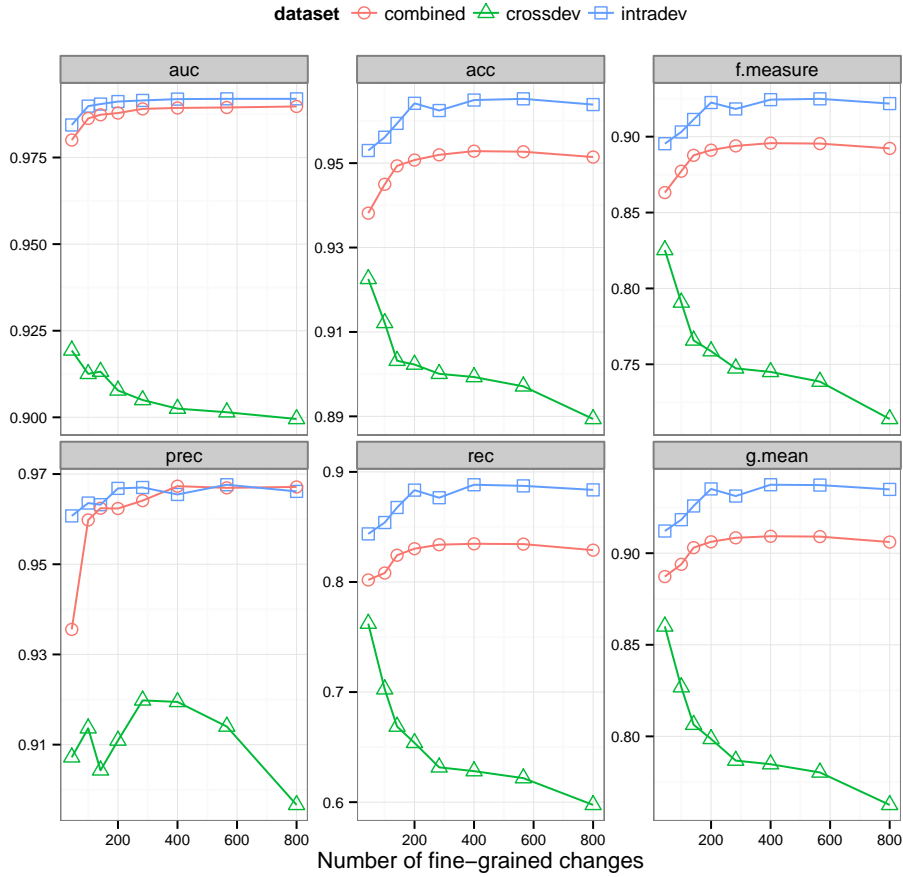


Figure 4.6: Dataset performance metrics

good when training with data from one developer and testing on the other (the *crossdev* dataset); moreover we see that as we increase the training size, there is a drop in performance. This can be attributed to overfitting the model to the working habits of each individual developer. Finally, we see that we can train accurate models by combining data from multiple developers. In the *combined* dataset, we combine the data generated by both developers and use this to train the model; this means that training and testing data is taken from both samples. Figure 4.6 shows that this dataset reaches high and stable results; and overfitting seems not present.

What is interesting to note is that the number of fine-grained changes required for training in both the *combined* and *intradev* cases is low: with 200 changes we can obtain prediction results only 2% worse (in terms of *acc*) on average than if we train

with 800 changes. As 200 fine-grained changes are the equivalent of a few days of work,³ we have encouraging evidence that an accurate model can be trained fast and deliver good results for a single developer. Moreover, a pre-trained model with data from multiple developers might be enough as a starting point for an untangling tool, which could then be trained to a particular developer’s working habits.

Overall, the results show that using the random forest algorithm, a randomized set of about 200 fine grained changes and a few easy-to-calculate voters, we can train a prediction model that can identify whether two changes belong in the same commit with an accuracy of 95% for a single developer.

4.4.3 How Effective Is EPICEAUNTANGLER for Developers?

We answer research question three by deploying EPICEAUNTANGLER with developers and recording whether the clustering that it proposes corresponds to participants’ expectations. The dataset devEval, resulting from this evaluation is described in Table 4.5. We notice that not all the developers coded full time during the two weeks, thus some produced fewer changes.

Table 4.5: Descriptive statistics of dataset devEval

P_ID	Total number of		Changes per cluster			
	changes	clusters	Mean	Median	St. Dev.	Max
P3	350	22	15.9	11	13.5	42
P4	826	28	29.5	3.5	50.9	228
P5	200	13	15.4	10	17.3	65
P6	166	12	13.8	6.5	15.6	47
P7	347	18	19.3	7	27.8	88
P8	162	11	14.7	10	12.7	37

We compared each cluster we proposed to the cluster that the participant eventually judged as correct to be committed. The histogram in Figure 4.7 shows the frequency of the obtained results: We observed a median⁴ success rate of 0.915 and an average of 0.753 with a standard deviation of 0.30. By inspecting the instances with a success rate in the range [0,0.4] we could not pinpoint any systematic error; we plan to further address these cases in future work.

We asked developers their opinion on the tool and received diverse feedback. Most developers were positive [P3, P4, P6, P8], *e.g.*, P3 expressed the feeling that

³From the data we recorded, 200 fine-grained code changes correspond to two to five days of work, depending on the developer’s style and pace.

⁴The results are not normally distributed, thus we report the median value.

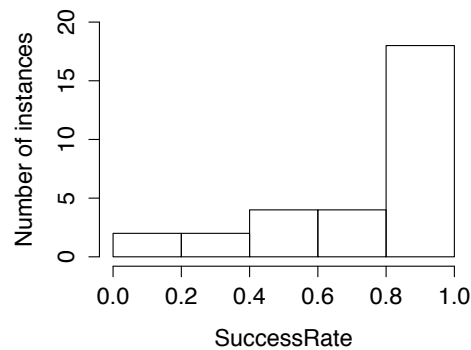


Figure 4.7: Frequency of success rate of EPICEA_{UNTANGLER} clustering approach

“EPICEA_{UNTANGLER} guesses correctly the clusters of changes, also in a big commit were I had 10 different clusters,” and P4 said: “It works good in many cases, especially for not so big change sets.” At the same time, most developers [P4, P5, P6, P8] expressed concerns with the large amount of fine-grained information to be processed; they explained that it adds too much noise to see not only the last state of a method but also all the intermediate modifications to it, especially when belonging to the same cluster. In the words of P8: “It was a bit painful to check everything.”

Some participants suggested improvements to the user interface: For example, P7 said that he “would like to option to delete tasks in the UI”, and P6 said: “I would like to type a name for each task in the UI, as a reminder while I cluster.”

4.5 Discussion

In this section we discuss our results and show how we mitigate the threats that endanger them.

4.5.1 Results

In the first research question, we asked which voters, or features are significant to untangle fine-grained code changes. Despite the fact that we implemented voters along six dimensions, only two dimensions were significant and contributed to most of the outcome: *code structure* and *spread*. In particular, the latter dimension has the greatest impact, by a large margin; the only significant voter in the former dimension measured whether the two changes were happening in the same class. This implies that these voters can be applied to other object-oriented programming languages regardless of whether they use types or not. This is a ripe opportunity for testing the approach with different languages and in different settings. Moreover, although we have no information about the significance of the voters

implemented by Herzig and Zeller [Herzig 2011, Herzig 2013], studies can be designed and carried out to determine if and how untangling effectiveness increases as a result of combining their voters with our significant ones.

We were surprised by the low impact of many of the voters in the untangling task: We expected message sending and variable access, as well as testing information, to contribute more. Since our initial data analysis was conducted with changes collected by only two developers, a further study with a larger set of developers for generating training data would be useful to confirm or alter this result.

In the second research question, we asked how effective is the best performing machine learning algorithm (*i.e.*, random forests) when used with the most significant voters. The results were overall very good. Expectedly, we achieved the best results when training and testing on data from the same developer, nevertheless cross-developers results are promising and merged-developers results do not show overfitting; in addition, approximately 200 fine-grained code changes were enough to reach most of the effectiveness. This implies that training on more developers is necessary to achieve a more general approach, and there seem to be no risk of overfitting by doing it. Moreover, ideally every user should train the approach on her own programming behavior; this seems reasonable since the training is effective with as little as a few days of work.

In the third research question, we investigate the effectiveness of the whole approach when deployed to programmers. Considering that the recruited participants were not used for the training phase, the results are in line with the effectiveness measured for RQ2. One of the most recurring complaints was about the large number of changes to be verified and sorted at every commit. This is due to the fact that we showed all the fine-grained changes recorded, thus also intermediate states for the same method (when the developer saved multiple times). We expect this information overload problem to be mitigated once the approach is stable enough to work correctly in most cases. Nevertheless, we see a good opportunity for further investigating how certain fine-grained code changes can be omitted, without losing relevant information that would lead to the incompleteness discussed in Section 4.1.1. Moreover, valuable comments were provided about the UI of EPICEAUNTANGLER. The UI evaluation goes beyond the scope of this chapter, but improving the UI can help to have an impact on reducing the information overload of fine-grained code changes.

4.5.2 Threats to Validity

Internal Validity. Our models and feature selection process are based on a dataset generated through the actions of two developers. While we have combined the actions of the developers and shown that they provide very good prediction per-

formance and the evaluation of the EPICEAUNTANGLER has been overwhelmingly positive, it is possible that our findings are biased towards the two developers' working habits.

Bias with respect to developer working habits might also occur in our selection of evaluation subjects. To reduce this risk, we selected diverse developers, all of them working in different projects and even in different physical locations. Thus, we believe the participants represent a heterogeneous enough population of PHARO developers.

Construct Validity. The notion of task is ambiguous. In particular, each participant can interpret the task granularity differently. For example, consider a single bug fix which is intended to fix two broken features. The participant could consider the changes either as two individual tasks, or everything as a single bug-fixing task. For mitigating this risk, we prepared a screencast with an example for users trying to establish a common criterion for task granularity. Moreover, we kept in close contact with users for answering any doubt. However, this ambiguity in the definition of task does not reduce the precision of our success metric for answering RQ3 (*SuccessRate*), since it represents each user expectation: it compares EPICEAUNTANGLER's clustering with the participant's expected clustering.

The clustering computed by EPICEAUNTANGLER may have influenced participants. When users had to evaluate the computed clustering (as shown in Figure 4.2), the initial clustering might have biased their answers.

External Validity. We used a specific platform (PHARO) and language environment (SMALLTALK) to facilitate our study. A specific language may dictate a specific working style. For example, in a typed language setting, an IDE would immediately mark as erroneous cases where a type signature has changed and not all uses have been adapted, therefore prompting the developer to fix such cases. Therefore our results may not be generalizable to all languages or working environments.

4.6 Related Works

The impact of tangled changes has been reported in several contexts: The inspiring work by Herzig and Zeller [Herzig 2013], reported that at least 16.5% of all source files in the datasets they considered were incorrectly associated with bug reports when ignoring the existence of tangled change sets. In a large-scale study done at Microsoft on how developers understand code changes, Tao *et al.* reported that developers find it important for understanding to decompose changes into the individual development issues, but there is currently no tool support for doing so [Tao 2012]. Bacchelli and Bird reported that tangled changes in code to be reviewed often cause low quality reviews or require longer time to review [Bacchelli 2013].

Herzig and Zeller were the first to implement an algorithm to automatically generate untangled commits given a tangled one. Their work greatly inspired our research. However, we see some limitations to their work that we explained in Section 4.1: static-analysis dependency, incompleteness, and artificiality. The main differences with our work is that:

1. we count with fine-grained timing information of code changes as well as IDE events like test runs;
2. we work in a dynamically-typed language;
3. we evaluated our approach with developers.

Another source of inspiration comes from Robbes, who created a fine-grained change model of software evolution based on three principles [Robbes 2008a]:

1. a program state needs to be represented accurately by an Abstract Syntax Tree (AST);
2. a program's history is a sequence of changes, each one producing a program state (an AST) and changes can be composed into higher-level changes;
3. changes should be recorded by the IDE as they happen, not recovered from a VCS.

Robbes *et al.* show how a fine-grained change model can better detect *logical coupling* between classes [Robbes 2008c]. Their chapter presents new measures of logical coupling that we consider as a future extension of our voters.

Steinert *et al.* propose CoEXIST, an approach and associated tool set to navigate the different states of a project based on its fine-grained changes [Steinert 2012]. CoEXIST's tool suite allows for reverting any fine-grained change at the project level, comparing different states of a program, localizing the cause of a failing test in the development history, and reassembling changes to share untangled commits. Automatic clustering of dependent fine-grained changes to create untangled commits is left as future work. Our work can be seen as an extension of CoEXIST tool suite in this direction, despite its totally unrelated implementation.

Wloka *et al.* presented a program analysis technique to identify commit-table changes that can be released early, without causing failures of existing tests [Wloka 2009]. Wloka remarks that an untangling algorithm would clearly benefit from having a model with a more accurate concept of change to add context information for individual change operations. Beyond our Same Test Run voter, we leverage more the results of unit-test execution to cluster related changes.

4.7 Conclusion

In this chapter, we have devised and evaluated `EPICEAUNTANGLER`, an approach whose ultimate goal is to help developers share self-contained changes that are well-decomposed into individual tasks. We build on the shoulders of others, and expand previous work by:

1. Working in an untyped language setting where static code analyses are more limited;
2. considering fine-grained code change information gathered during development; and
3. evaluating the resulting approach both on data generated by programmers who manually labeled it and with programmers working on real development tasks.

Our results show that three features are especially important to perform clustering of fine-grained code changes: the time between the changes, the number of other modifications between the changes, and whether the changes modify the same class. By testing the features on historical data manually labeled by developers, we obtained good results (over 88% of accuracy in the worst case) in determining whether two changes should be together. When deploying our approach with new developers, we obtained a median success rate of 91%.

Overall, this chapter makes the following main contributions:

1. An analysis of the current points for improvement in the state of the art in untangling code changes.
2. A publicly available⁵ dataset of fine-grained code changes collected by recording the development sessions of two developers over the course of four months, and the corresponding manual clustering.
3. The creation of different features/voters and their evaluation, based on the aforementioned dataset, using machine learning approaches to model and classify pairs of fine-grained code changes, resulting in good accuracy results.
4. The creation of an approach, `EPICEAUNTANGLER`, and corresponding tool implementation, `EPICEA TASK CLUSTERER`,⁶ to untangle fine-grained code changes into clusters based on the three best voters and the best performing machine learning algorithm.
5. The deployment and a two-week evaluation of `EPICEAUNTANGLER` with developers with good results.

⁵Available at: <http://dx.doi.org/10.6084/m9.figshare.1241571>

⁶Available at: <http://smalltalkhub.com/#!/~MartinDias/EpiceaTaskClusterer>

5 CONCLUSION AND FUTURE WORK

Contents

5.1 Summary	63
5.2 Future Work	65
5.3 Publications	69

5.1 Summary

During its lifetime, a system's codebase changes due to concerns such as bug fixes, new features, and migrations. According to best practices, the implementation of such concerns happen in isolation from each others, to avoid potential interferences while they are not ready. This means that codebases temporarily diverge during development and must be integrated back when finished.

In the development team of a software system, numerous developers may collaborate in the evolution of the system's codebase. Integrators have the crucial role of integrating developer's code changes. The integration activities require understanding somebody else's changes, to be integrated in a codebase that is often different than the original one where the developer worked. Due to the complexity of integration, developers need as much help as they can get from tools.

In our thesis we claimed that the reification of domain entities of software evolution provides a more comprehensive support to integration activities.

Chapter 2. The ultimate goal of our work is to tackle real-world integration problems by proposing new tools. To that end, we wanted to understand what are the key integration activities without comprehensive tool support in real-world integrators. We found that there is no quantitative information to this respect in literature. Thus, we ran a survey to rank the importance of integration activities and the level of tool support for each. We analyzed the collected survey responses, and identified key integration activities without comprehensive tool support: From the analysis of the collected survey responses, we identified three key integration activities without comprehensive tool support: *understanding change scattering*; *understanding change dependencies when cherry-picking*; and *understanding change impact*. This list of integration activities are insights that serve as guidelines to direct our efforts on new approaches to improve everyday's work of integrators.

Chapter 3. In this chapter we analyzed the state of the art in VCS and software evolution approaches. We concluded that reification of main concepts can reduce the semantic gap present in nowadays' VCSs and thus improve integration activities. Next, we presented EPICEA, a first-class change model which includes: *low-level code changes* such as *class addition* and *method modification*; *high-level code changes* such as the *method rename* refactoring; *IDE interaction data*, including extra developer's actions in the IDE such as *unit test run*. This change model requires information that can be gathered from the developer when working, and is hard to reconstruct from VCS data. Thus, we implemented an EPICEA monitor, that listens developer actions from the PHARO IDE and records EPICEA model instances in a data base. We reported on the implementation of EPICEA model and associated tools, that allowed us to evaluate our approaches in real-world scenarios.

Chapter 4. In this chapter we focused in *understanding change scattering*, one of the key integration activities we identified from survey responses. We reported on a novel approach, EPICEAUNTANGLER, to help developers share untangled commits (aka. atomic commits) by using fine-grained code change information gathered from the IDE through EPICEA model and tools. Our results showed that three features are especially important to perform clustering of fine-grained code changes: the time between the changes; the number of other modifications between the changes; and whether the changes modify the same class. By testing the features on fine-grained code changes manually labeled by developers, we obtained good results (over 88% of accuracy in the worst case) in determining whether two changes should be together. When deploying our approach with new developers, we obtained a median success rate of 91%. Additionally, we published the aforementioned dataset of fine-grained code changes manually labeled by developers, so other researchers can access it.

Conclusion. We argued in Chapter 3 and concluded in Chapter 4 that gathering *interaction data* from developer's IDE provides more complete and precise information to allow a new generation tools. Moreover, the outcomes of this work indicate that existing integration process can benefit from tools that explicitly model code changes and gather developers' activities performed in the IDEs.

Additionally, the general concepts of the approaches proposed in this work are independent of the underlying programming language. Then, nothing limits our approaches to be applied to other dynamically-typed languages such as JAVASCRIPT, RUBY and PYTHON, neither to statically-typed languages such as JAVA.

Nevertheless, we believe PHARO was a good choice to implement and evaluate our approaches. In that regards, the following section outlines future research plans using EPICEA model of developer's activities in the IDE.

5.2 Future Work

Mining Software Repositories (MSR) [Herzig 2010] is an empirical research field that exploits the information developers produce, that is stored in software repositories (such as a VCS or an issue tracking system). We can extract from the results of this thesis that gathering interaction data in first-class entity models can contribute to this research field, and also allow a new generation of development tools.

We are currently discussing with leaders in the Pharo community how to integrate EPICEA in the next official version of PHARO. This will allow us to collect interaction data about code changes and IDE interactions from participants doing real-world development work. This will also enable us to deploy additional tools with more participants to evaluate their results. This is important to us, since we want both to improve the state of the art and to create approaches that can be used in real-world scenarios.

In the following, we describe future research tracks based on EPICEA's interaction data that deal with two pending challenges to tackle:

Unexplored case studies. Besides techniques of mining developer interaction data have been successful in valuable software engineering problems, there are still many problems that have not been explored. In particular, we focus on two use cases for interaction data: code review, and mitigating ripple effects.

Scarce public data. Public interaction data is scarce. One of the contributions of our work is the EPICEAUNTANGLER dataset. This dataset of untangled code changes was created with the help of two developers who accurately split their code changes into self contained tasks over a period of four months. Nevertheless, few other public datasets exist. EPICEA will be enabled by default for the PHARO open-source community in the next PHARO major release. This will facilitate the generation of additional interaction datasets, a valuable resource to evaluate our work.

Mining interaction data can be useful in a large number of software engineering research areas. Following, we devise future work revolved around two valuable software engineering and evolution problems: first, to improve modern code review by facilitating change untangling and change understanding; and second, to mitigate ripple effects by facilitating the generation of evolution rules and increasing awareness. We believe these problems are good case studies for building and evaluating new approaches based on mining developer interaction data. Additionally, we can also generate new interaction datasets and publish them as a contribution to the scientific community. This is important given the current

scarcity of public interaction datasets, which is a roadblock to further software engineering research in the area.

5.2.1 Interaction Data to Improve Modern Code Review

Modern Code Review (MCR) [Cohen 2010] is an important mechanism for quality assurance in software evolution: for example, it provides feedback and helps to avoid introducing bugs. MCR is a lightweight variant of the code inspections investigated since the 1970s, which prevails today both in industry and open-source software (OSS) systems [Beller 2014]. Large companies and OSS communities such as Microsoft, Google and Linux use MCR as a regular practice [Bosu 2015, Bird 2015]. The first challenge of MCR is code understanding [Bacchelli 2013], because it is time consuming and demands a big effort from the reviewers. This challenge is accentuated when the change sets to examine is large [Mockus 2002, Rigby 2012]. In this track we want to investigate how MCR can be improved by interaction data.

We have some ideas to develop and evaluate:

Assisting commit descriptions. When a developer wants to commit local code changes to the VCS repository, typically he or she writes a description summarizing the changes [Rigby 2008]. When this happens after a long coding session, such task can be tedious, and result in bad-quality descriptions (too coarse grained or inaccurate). A bad description hinders the understanding of the code changes to the reviewers. In this scenario, an IDE plugin can monitor interaction data to offer the developer high-level descriptions (*e.g.*, refactoring) at the moment of commit.

We foresee some challenges in this approach: First, many changes are shadowed or undone and thus do not reach the VCS [Negara 2012]. Second, detection of high-level changes is not trivial. For example, developers often do refactorings manually even if the IDE offers automatic refactorings [Negara 2013].

Labelling code changes in the reviewing tool. Going a step further from the previous point, the high-level descriptions of the code changes can be exported to the reviewing tool as metadata. By means of such metadata, a code reviewing tool can use specialized views to ease the work of reviewers. Zhang *et al.* built and evaluated an approach where reviewers could interactively apply high-level templates to summarize similar changes and detect potential mistakes, showing good results [Zhang 2015].

Tao *et al.* reported a work on change untangling in the context of code review [Tao 2015]. Their MSR approach exploits information in the VCS repository to present the code changes to the reviewers in slices. The goal is to help

the reviewer to understand such code changes. We want to use our experience in untangling code changes using EPICEA fine-grained code changes and interaction data to contribute in this line of work.

Intra-session author explanations. Sometimes, code changes include workarounds or odd code which is hard to understand. When a developer performs a code change in a system that follows a MCR process, he is conscious that the change will be reviewed afterwards. Since the coding session before the commit to VCS can be long, ideally the author of the change will explain his or her odd code change just after doing it, because it's fresh. However, with existing tools, a developer can only explain the change after the coding session is finished. We believe that an IDE plugin extending EPICEA tools can allow the developer to explain changes in midst of the coding session.

In future work, we plan to develop approaches for these ideas and conduct controlled experiments to explore and validate them. Afterwards, the lessons learnt will serve us to evaluate such approaches in real-world settings to learn how feasible they are with real developers.

5.2.2 Interaction Data to Mitigate Ripple Effects

Often, when the codebase of a system changes, other systems that depend on it need to be migrated. This change propagation is known as ripple effect [Yau 1978]. Ripple effects require constant attention: a small change in a system can have a very large impact on other systems. Additionally, sometimes a code that needs migration remains undiscovered for a long time. The author of code changes sometimes writes migration instructions in a change log. However, migration instructions are not useful in many cases. Robbes *et al.* found that in nearly 40% of the studied cases the migration instructions are either absent, unclear, or the developers decide not to take into account the advice that they received [Robbes 2012].

Interaction data to assist code migration. Hora *et al.* worked on the extraction of migration rules from VCS repositories [Hora 2013, Hora 2014]. The key idea is that developers can use such system-specific rules as guides to migrate their systems according to changes in depended systems. From our point of view, the use of VCS as a data source is a limitation that IDE integration and interaction data can overcome: We described above a future approach where the developer could tag code changes with high-level labels during the coding session as an extended explanation for the reviewers. In this case, we propose to provide the developer richer labels for code changes: labels that can guide other developers to migrate their systems in consequence of the change. For example, a developer that removes a class can label such change as a deprecation and specify other classes that work

as a replacement. In this case, a label is a migration rule associated with a specific commit. Then, the migration from one version to another of some system can be assisted by applying all the associated migration rules between such versions. Our hypothesis is that an approach that uses interaction data can assist the author of changes to create migration rules with low effort. To evaluate this hypothesis, we believe that it's need to perform some exploratory studies, followed by some controlled experiments, to finally experiment in real-world use cases.

Early detection of conflicts: workspace awareness. The *commit-based* property of existing VCSs (presented in Chapter 3) promote development workspaces that isolate developers from each other. This isolation has pros and cons: it is good, because developers make their changes without any interference from changes made concurrently by other developers; and it is bad, because not knowing which artifacts are changing in parallel regularly leads to problems when changes are promoted from workspaces into a VCS repository. Overcoming the bad isolation, while retaining the good isolation, is a matter of raising *awareness* among developers. A number of *workspace awareness* techniques have been proposed to enhance the effectiveness of VCSs in coordinating parallel work [Hattori 2010, Holmes 2010, Guimarães 2012, Sarma 2014]. These techniques share information regarding ongoing changes, so potential conflicts can be detected during development, instead of when changes are completed and committed to a VCS repository. We think workspace awareness is a promising case study where to apply interaction data lessons learnt during our research.

5.3 Publications

Following, we list the papers published during this Ph.D thesis.

Untangling Fine-Grained Code Changes

Abstract: *After working for some time, developers commit their code changes to a version control system. When doing so, they often bundle unrelated changes (e.g., bug fix and refactoring) in a single commit, thus creating a so-called tangled commit. Sharing tangled commits is problematic because it makes review, reversion, and integration of these commits harder and historical analyses of the project less reliable. Researchers have worked at untangling existing commits, i.e., finding which part of a commit relates to which task. In this paper, we contribute to this line of work in two ways: (1) A publicly available dataset of untangled code changes, created with the help of two developers who accurately split their code changes into self contained tasks over a period of four months; (2) a novel approach, EPICEAUNTANGLER, to help developers share untangled commits (aka. atomic commits) by using fine-grained code change information. EPICEAUNTANGLER is based and tested on the publicly available dataset, and further evaluated by deploying it to 7 developers, who used it for 2 weeks. We recorded a median success rate of 91% and average one of 75%, in automatically creating clusters of untangled fine-grained code changes.*

Authors: Martín Dias and Alberto Bacchelli and Georgios Gousios and Damien Cassou and Stéphane Ducasse.

Venue: SANER'15: 22nd International Conference on Software Analysis, Evolution, and Reengineering.

Note: Candidate for IEEE Research Best Paper Award.

URL: <https://hal.inria.fr/hal-01116225>

DELTAIMPACTFINDER: Assessing Semantic Merge Conflicts with Dependency Analysis

Abstract: *In software development, version control systems (VCS) provide branching and merging support tools. Such tools are popular among developers to concurrently change a codebase in separate lines and reconcile their changes automatically afterwards. However, two changes that are correct independently can introduce bugs when merged together. We call semantic merge conflicts this kind of bugs. Change impact analysis (CIA) aims at estimating the effects of a change in a codebase. In this paper, we propose to detect semantic merge conflicts using CIA. On a merge, DELTAIMPACTFINDER analyzes and compares the impact of a change in its origin and destination branches. We call the difference between these two impacts the delta-impact. If the delta-impact is empty, then there is no indicator of a semantic merge conflicts and the merge can continue automatically. Otherwise, the delta-impact contains what are the sources of possible conflicts.*

Authors: Martín Dias and Guillermo Polito and Damien Cassou and Stéphane Ducasse

Venue: International Workshop on Smalltalk Technologies, Brescia, Italy, 2015.

URL: <https://hal.inria.fr/hal-01199035>

Representing Code History with Development Environment Events

Abstract: *Modern development environments handle information about the intent of the programmer: for example, they use abstract syntax trees for providing high-level code manipulation such as refactorings; nevertheless, they do not keep track of this information in a way that would simplify code sharing and change understanding. In most Smalltalk systems, source code modifications are immediately registered in a transaction log often called a ChangeSet. Such mechanism has proven reliability, but it has several limitations. In this paper we analyze such limitations and describe scenarios and requirements for tracking fine-grained code history with a semantic representation. We present Epicea, an early prototype implementation. We want to enrich code sharing with extra information from the IDE, which will help understanding the intention of the changes and let a new generation of tools act in consequence.*

Authors: Martín Dias and Damien Cassou and Stéphane Ducasse.

Venue: International Workshop on Smalltalk Technologies, Annecy, France, 2013.

URL: <https://hal.inria.fr/hal-00862626>

Software Integration Questions: A Quantitative Survey

Abstract: *Software is in constant evolution. In a software project, code changes represent bug fixes, enhancements, new features and adaptations due to changing domains. The evolution of a project codebase is usually managed in a revision control system that supports branches. Developers perform code changes in a branch and sometimes such changes are merged into other branch. This activity is called integration. Integration of changes poses substantial challenges. We conducted a survey to evaluate a catalogue of 46 questions about integration. For each question, the participants had to rank the importance and the support that current tools offer. In a period of 5 months we received the responses of 42 developers who integrate changes on very diverse software projects.*

Authors: Martín Dias and Verónica Uquillas-Gomez and Damien Cassou and Stéphane Ducasse

Venue: No (Technical Report)

URL: <https://hal.inria.fr/hal-01093496>

Do Tools Support Code Integration? A Survey

Abstract: *Integrating changes made by other developers is a difficult and tedious process. To understand how to help integrators, we first need to know the main questions they ask themselves while integrating and then relate these questions to the tool support that is needed. With this information, researchers and tool developers will be able to focus on the important questions that have little tool support. In this paper, we report on a 2-step study. In the first step, we did an open call to integrators. We ask them to list questions they ask themselves when they integrate a change. In the second step, based on the questions gathered during the first step and a literature survey, we built a list of 46 questions and run a survey to rank the importance of each question and if the level of tool support was adequate. We present the results we collected. Additionally, we present a taxonomy of the questions according to the kind of information that tools need to answer such questions. We found out that some questions like “Who is the author of this changed code?” are important and have good tool support whereas others like “Do all the changes within the commit belong together? (Can we split the commit?)” are moderately to extremely important and have no tool support.*

Authors: Martín Dias and Damien Cassou and Stéphane Ducasse and Verónica Uquillas-Gomez

Venue: The Journal of Object Technology (under review)

URL: Not available (request if wanted)

Bibliography

- [Bacchelli 2013] Alberto Bacchelli and Christian Bird. *Expectations, Outcomes, and Challenges of Modern Code Review*. In Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pages 712–721, Piscataway, NJ, USA, 2013. IEEE Press. 39, 60, 66
- [Beck 2000] Kent Beck. *Extreme programming explained: Embrace change*. Addison Wesley, 2000. 39
- [Beller 2014] Moritz Beller, Alberto Bacchelli, Andy Zaidman and Elmar Juergens. *Modern Code Reviews in Open-source Projects: Which Problems Do They Fix?* In Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, pages 202–211, New York, NY, USA, 2014. ACM. 66
- [Bird 2015] Christian Bird, Trevor Carnahan and Michaela Greiler. *Lessons Learned from Building and Deploying a Code Review Analytics Platform*. In Proceedings of the International Conference on Mining Software Repositories. IEEE, 2015. 66
- [Black 2009] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou and Marcus Denker. *Pharo by example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009. 5
- [Bosu 2015] Amiangshu Bosu, Michaela Greiler and Christian Bird. *Characteristics of Useful Code Reviews: An Empirical Study at Microsoft*. In Proceedings of the International Conference on Mining Software Repositories. IEEE, 2015. 66
- [Breiman 2001] Leo Breiman. *Random forests*. *Machine learning*, vol. 45, no. 1, pages 5–32, 2001. 39, 46
- [Chawathe 1996] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina and Jennifer Widom. *Change Detection in Hierarchically Structured Information*. *SIGMOD Rec.*, vol. 25, no. 2, pages 493–504, June 1996. 27
- [Cohen 2010] Jason Cohen. *Modern code review*, chapitre 18. O'Reilly Media, Inc., October 2010. 66
- [Demeyer 2000] Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz. *Finding Refactorings via Change Metrics*. In Proceedings of 15th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00), pages 166–178, New York NY, 2000. ACM Press. Also in ACM SIGPLAN Notices 35 (10). 28
- [Demeyer 2002] Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz. *Object-oriented reengineering patterns*. Morgan Kaufmann, 2002. 1, 7

- [Dias 2014] Martín Dias, Verónica Uquillas-Gomez, Damien Cassou and Stéphane Ducasse. *Software Integration Questions: A Quantitative Survey*. Rapport technique, INRIA Lille, 2014. 11, 15
- [Dig 2006] Danny Dig, Can Comertoglu, Darko Marinov and Ralph Johnson. *Automated Detection of Refactorings in Evolving Components*. In ECOOP, pages 404–428, 2006. 28
- [Dig 2008] Danny Dig, Kashif Manzoor, Ralph E. Johnson and Tien N. Nguyen. *Effective Software Merging in the Presence of Object-Oriented Refactorings*. IEEE Transactions on Software Engineering, vol. 34, no. 3, pages 321–335, 2008. 28
- [Ducasse 2000] Stéphane Ducasse, Michele Lanza and Sander Tichelaar. *Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems*. In Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools, CoSET '00, June 2000. 27
- [Ebraert 2007] Peter Ebraert, Jorge Vallejos, Pascal Costanza, Ellen Van Paesschen and Theo D'Hondt. *Change-oriented software engineering*. In Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference, ICDL '07, pages 3–24. ACM, 2007. 28
- [Eick 2001] Stephen Eick, Todd Graves, Alan Karr, J. Marron and Audris Mockus. *Does Code Decay? Assessing the Evidence from Change Management Data*. IEEE Transactions on Software Engineering, vol. 27, no. 1, pages 1–12, 2001. 25
- [Falleri 2014] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez and Martin Montperrus. *Fine-grained and Accurate Source Code Differencing*. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, pages 313–324, New York, NY, USA, 2014. ACM. 27
- [Fluri 2007] Beat Fluri, Michael Wuersch, Martin Pinzger and Harald Gall. *Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction*. IEEE Transactions on Software Engineering, vol. 33, no. 11, pages 725–743, 2007. 27
- [Foster 2012] Stephen R. Foster, William G. Griswold and Sorin Lerner. *WitchDoctor: IDE Support for Real-time Auto-completion of Refactorings*. In Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pages 222–232, Piscataway, NJ, USA, 2012. IEEE Press. 29
- [Fowler 1999a] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999. 39

- [Fowler 1999b] Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts. *Refactoring: Improving the design of existing code*. Addison Wesley, 1999. ordered but not received. 2
- [Fritz 2010] Thomas Fritz and Gail C. Murphy. *Using information fragments to answer the questions developers ask*. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE'10, pages 175–184. ACM, 2010. 7, 10
- [Gall 2009] H C Gall, B Fluri and M Pinzger. *Change analysis with evolizer and ChangeDistiller*. IEEE Software, vol. 26, no. 1, pages 26–33, feb 2009. 27
- [Ge 2012] Xi Ge, Quinton L. DuBose and Emerson Murphy-Hill. *Reconciling Manual and Automatic Refactoring*. In Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pages 211–221, Piscataway, NJ, USA, 2012. IEEE Press. 29
- [Genuer 2010] Robin Genuer, Jean-Michel Poggi and Christine Tuleau-Malot. *Variable selection using random forests*. Pattern Recognition Letters, vol. 31, no. 14, 2010. 54
- [Gırba 2005] Tudor Gırba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Bern, Bern, November 2005. 27
- [Gırba 2006] Tudor Gırba and Stéphane Ducasse. *Modeling History to Analyze Software Evolution*. Journal of Software Maintenance: Research and Practice (JSME), vol. 18, pages 207–236, 2006. 27
- [Goldberg 1989] Adele Goldberg and Dave Robson. *Smalltalk-80: The language*. Addison Wesley, 1989. 5
- [Gousios 2014] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey and Arie van Deursen. *Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective*. Rapport technique 013, Technical University Delft - SERG, 2014. 7, 8, 39
- [Guimarães 2012] Mário Luís Guimarães and António Rito Silva. *Improving Early Detection of Software Merge Conflicts*. In Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pages 342–352, Piscataway, NJ, USA, 2012. IEEE Press. 68
- [Guzzi 2015] Anja Guzzi, Alberto Bacchelli, Yann Riche and Arie van Deursen. *Supporting Developers' Coordination in the IDE*. In In Proceedings of CSCW 2015 (8th ACM Conference on Computer Supported Cooperative Work and Social Computing), page in press. ACM, 2015. 37, 39
- [Hastie 2001] Trevor Hastie, Robert Tibshirani and Jerome Friedman. *The elements of statistical learning*. Springer, 2001. 46, 50
- [Hattori 2010] L. Hattori and M. Lanza. *Syde: a tool for collaborative software development*. In ICSE Tool demo, pages 235–238. ACM, 2010. 42, 68

- [Herzig 2010] Kim Herzig and Andreas Zeller. Mining your own evidence, chapitre 27. O'Reilly Media, Inc., October 2010. 25, 65
- [Herzig 2011] Kim Herzig and Andreas Zeller. *Untangling changes*. Unpublished manuscript, September 2011. 37, 38, 40, 42, 59
- [Herzig 2013] Kim Herzig and Andreas Zeller. *The Impact of Tangled Code Changes*. In Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, pages 121–130, Piscataway, NJ, USA, 2013. IEEE Press. 23, 25, 38, 39, 40, 42, 59, 60
- [Holmes 2010] Reid Holmes and Robert J. Walker. *Customized Awareness: Recommending Relevant External Change Events*. In Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, pages 465–474, New York, NY, USA, 2010. ACM. 68
- [Hora 2013] Andre Hora, Nicolas Anquetil, Stéphane Ducasse and Marco Túlio Valente. *Mining System Specific Rules from Change Patterns*. In Proceedings of the 20th Working Conference on Reverse Engineering (WCRE'13), 2013. 25, 67
- [Hora 2014] Andre Hora, Anne Etien, Nicolas Anquetil, Stéphane Ducasse and Marco Túlio Valente. *APIEvolutionMiner: Keeping API Evolution under Control*. In Proceedings of the Software Evolution Week (CSMR-WCRE'14), 2014. 5, 25, 67
- [Karypis 1995] George Karypis and Vipin Kumar. *Analysis of Multilevel Graph Partitioning*. In Proceedings of Supercomputing 1995 (ACM/IEEE Conference on Supercomputing). ACM, 1995. 40
- [LaToza 2010] Thomas D. LaToza and Brad A. Myers. *Hard-to-answer questions about code*. In Evaluation and Usability of Programming Languages and Tools, PLATEAU 10, pages 8:1–8:6. ACM, 2010. 7, 8, 10
- [Laval 2009] Jannik Laval, Simon Denier, Stéphane Ducasse and Andy Kellens. *Supporting Incremental Changes in Large System Models*. In Proceedings of ESUG International Workshop on Smalltalk Technologies, IWST'09, pages 1–7, Brest, France, 2009. 27
- [Laval 2011] Jannik Laval, Simon Denier, Stéphane Ducasse and Jean-Rémy Falleri. *Supporting Simultaneous Versions for Software Evolution Assessment*. Journal of Science of Computer Programming (SCP), vol. 76, no. 12, pages 1177–1193, May 2011. 27
- [Lehman 1980] Manny Lehman. *Programs, Life Cycles, and Laws of Software Evolution*. Proceedings of the IEEE, vol. 68, no. 9, pages 1060–1076, September 1980. 1, 7
- [Lippe 1992] Ernst Lippe and Norbert van Oosterom. *Operation-based merging*. In Proceedings of the 5th ACM SIGSOFT symposium on Software Develop-

- ment Environments, SDE'92, pages 78–87, New York, NY, USA, 1992. ACM Press. 28
- [Maalej 2014] Walid Maalej, Thomas Fritz and Romain Robbes. *Collecting and Processing Interaction Data for Recommendation Systems*. In *Recommendation Systems in Software Engineering*, pages 173–197. Springer-Verlag, 2014. 28
- [Martinez 2014] Matias Martinez, Westley Weimer and Martin Monperrus. *Do the Fix Ingredients Already Exist? An Empirical Inquiry into the Redundancy Assumptions of Program Repair Approaches*. CoRR, vol. abs/1403.6322, 2014. 25
- [Mens 2002] T. Mens. *A State-of-the-Art Survey on Software Merging*. *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pages 449–462, 2002. 26
- [Mens 2005] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld and Mehdi Jazayeri. *Challenges in Software Evolution*. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE 2005)*, pages 123–131. IEEE Computer Society, 2005. 1
- [Mockus 2002] Audris Mockus, Roy T Fielding and James D Herbsleb. *Two case studies of open source software development: Apache and Mozilla*. *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 3, pages 309–346, 2002. 66
- [Nagappan 2005] Nachiappan Nagappan and Thomas Ball. *Use of Relative Code Churn Measures to Predict System Defect Density*. In *Proceedings of the 27th International Conference on Software Engineering, ICSE '05*, pages 284–292, New York, NY, USA, 2005. ACM. 25
- [Negara 2012] Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E. Johnson and Danny Dig. *Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?* In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP)*, 2012. 25, 26, 29, 39, 66
- [Negara 2013] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson and Danny Dig. *A Comparative Study of Manual and Automated Refactorings*. In *27th European Conference on Object-Oriented Programming*, pages 552–576, 2013. 66
- [Negara 2014] Stas Negara, Mihai Codoban, Danny Dig and Ralph E. Johnson. *Mining Fine-grained Code Changes to Detect Unknown Change Patterns*. In *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, pages 803–813, New York, NY, USA, 2014. ACM. 29
- [O’Neil 2013] Cathy O’Neil and Rachel Schutt. *Doing data science*. O’Reilly, 2013. 49
- [Phillips 2011] Shaun Phillips, Jonathan Sillito and Rob Walker. *Branching and Merging: An Investigation into Current Version Control Practices*. In *Proceedings of the 4th International Workshop on Cooperative and Human As-*

- pects of Software Engineering, CHASE '11, pages 9–15, New York, NY, USA, 2011. ACM. 8
- [Phillips 2012] Shaun Phillips, Guenther Ruhe and Jonathan Sillito. *Information needs for integration decisions in the release process of large-scale parallel development*. In Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work, CSCW'12, pages 1371–1380. ACM, 2012. 8
- [Premraj 2011] Rahul Premraj, Antony Tang, Nico Linssen, Hub Geraats and Hans van Vliet. *To branch or not to branch?* In Proceedings of the 2011 International Conference on Software and Systems Process, ICSSP'11, pages 81–90. ACM, 2011. 7, 9
- [Prete 2010] Kyle Prete, Napol Rachatasumrit, Nikita Sudan and Miryung Kim. *Template-based Reconstruction of Complex Refactorings*. In 26th International Conference on Software Maintenance, pages 1–10, 2010. 28
- [Renggli 2010] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba and Oscar Nierstrasz. *Domain-Specific Program Checking*. In Jan Vitek, editeur, Proceedings of the 48th International Conference on Objects, Models, Components and Patterns (TOOLS'10), volume 6141 of LNCS, pages 213–232. Springer-Verlag, 2010. 5
- [Rigby 2008] Peter C. Rigby, Daniel M. German and Margaret-Anne Storey. *Open Source Software Peer Review Practices: A Case Study of the Apache Server*. In Proceedings of the 30th International Conference on Software Engineering, ICSE '08, pages 541–550, New York, NY, USA, 2008. ACM. 66
- [Rigby 2012] Peter Rigby, Brendan Cleary, Frederic Painchaud, Margaret-Anne Storey and Daniel German. *Contemporary Peer Review in Action: Lessons from Open Source Development*. IEEE Softw., vol. 29, no. 6, pages 56–61, nov 2012. 66
- [Robbes 2005] Romain Robbes and Michele Lanza. *Versioning Systems for Evolution Research*. In Proceedings of IWPSE 2005 (8th International Workshop on Principles of Software Evolution), pages 155–164. IEEE Computer Society, 2005. 25, 26
- [Robbes 2007] Romain Robbes. *Mining a Change-Based Software Repository*. In Proceedings of the Fourth International Workshop on Mining Software Repositories, MSR'07, pages 15–, Washington, DC, USA, 2007. IEEE Computer Society. 42
- [Robbes 2008a] Romain Robbes. *Of Change and Software*. PhD thesis, University of Lugano, Switzerland, December 2008. 61
- [Robbes 2008b] Romain Robbes and Michele Lanza. *SpyWare: a change-aware development toolset*. In Proceedings of the 30th International Conference on

- Software Engineering, ICSE'08, pages 847–850, New York, NY, USA, 2008. ACM. 28
- [Robbes 2008c] Romain Robbes, Damien Pollet and Michele Lanza. *Logical Coupling Based on Fine-Grained Change Information*. In *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*, pages 42–46. IEEE, 2008. 61
- [Robbes 2012] Romain Robbes, Mircea Lungu and David Röthlisberger. *How Do Developers React to API Deprecation?: The Case of a Smalltalk Ecosystem*. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12*, pages 56:1–56:11, New York, NY, USA, 2012. ACM. 67
- [Sarma 2014] Anita Sarma, Josh Branchaud, Matthew B. Dwyer, Suzette Person and Neha Rungta. *Development Context Driven Change Awareness and Analysis Framework*. In *Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014*, pages 404–407, New York, NY, USA, 2014. ACM. 68
- [Schärli 2003] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz and Andrew P. Black. *Traits: Composable Units of Behavior*. In *Proceedings of European Conference on Object-Oriented Programming*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003. 32
- [Sillito 2005] Jonathan Sillito, Kris De Volder, Brian Fisher and Gail Murphy. *Managing software change tasks: An exploratory study*. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 23–32. IEEE Computer Society, 2005. 9
- [Sillito 2006] J. Sillito, G.C. Murphy and K. De Volder. *Questions Programmers Ask During Software Evolution Tasks*. In *Proceedings of the 14th International Symposium on Foundations on Software Engineering, SIGSOFT '06/FSE-14*, pages 23–34. ACM, 2006. 9
- [Sillito 2008] J. Sillito, G.C. Murphy and K. De Volder. *Asking and Answering Questions during a Programming Change Task*. *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pages 434–451, jul 2008. 7, 9, 10
- [Steinert 2012] Bastian Steinert, Damien Cassou and Robert Hirschfeld. *CoExist: Overcoming Aversion to Change - Preserving Immediate Access to Source Code and Run-time Information of Previous Development States*. In *DLS'12: Proceedings of the 8th Dynamic Languages Symposium, DLS '12*, pages 107–118, New York, NY, USA, 2012. ACM. 28, 39, 61
- [Tao 2012] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang and Sunghun Kim. *How Do Software Engineers Understand Code Changes?: An Exploratory Study in Industry*. In *Proceedings of FSE 2012 (20th ACM SIGSOFT Inter-*

- national Symposium on the Foundations of Software Engineering). ACM, 2012. 39, 60
- [Tao 2015] Yida Tao and Sunghun Kim. *Partitioning Composite Code Changes to Facilitate Code Review*. In Proceedings of the 12th Working Conference on Mining Software Repositories, MSR 2015, 2015. 66
- [Tichelaar 2000] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer and Oscar Nierstrasz. *A Meta-model for Language-Independent Refactoring*. In Proceedings of International Symposium on Principles of Software Evolution, ISPSE'00, pages 157–167. IEEE Computer Society Press, 2000. 27
- [Uquillas Gómez 2010a] Verónica Uquillas Gómez, Stéphane Ducasse and Theo D'Hondt. *Meta-models and Infrastructure for Smalltalk Omnipresent History*. In Smalltalks'2010, 2010. 27
- [Uquillas Gómez 2010b] Verónica Uquillas Gómez, Stéphane Ducasse and Theo D'Hondt. *Visually Supporting Source Code Changes Integration: the Torch Dashboard*. In Proceedings of the 17th Working Conference on Reverse Engineering (WCRE'10), pages 55–64, October 2010. 12
- [Uquillas Gómez 2012a] Verónica Uquillas Gómez. *Supporting Integration Activities in Object-Oriented Applications*. PhD thesis, Vrije Universiteit Brussel - Belgium & Université Lille 1 - France, October 2012. 7, 10, 11, 27, 39
- [Uquillas Gómez 2012b] Verónica Uquillas Gómez, Stéphane Ducasse and Theo D'Hondt. *Ring: a Unifying Meta-Model and Infrastructure for Smalltalk Source Code Analysis Tools*. Journal of Computer Languages, Systems and Structures, vol. 38, no. 1, pages 44–60, April 2012. 5, 27
- [Uquillas Gómez 2014] Verónica Uquillas Gómez, Stéphane Ducasse and Andy Kellens. *Supporting Streams of Changes during Branch Integration*. Journal of Science of Computer Programming, 2014. 14, 26, 27
- [v. Deursen 2008] A. v. Deursen, L. Moonen and A. Zaidman. Software evolution, chapitre 8: On the Interplay Between Software Testing and Evolution and its Effect on Program Comprehension. Springer, 2008. 1
- [Verwaest 2011] Toon Verwaest, Camillo Bruni, Mircea Lungu and Oscar Nierstrasz. *Flexible object layouts: enabling lightweight language extensions by intercepting slot access*. In Proceedings of 26th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11), pages 959–972, New York, NY, USA, 2011. ACM. 5
- [Weissgerber 2006] Peter Weissgerber and Stephan Diehl. *Identifying Refactorings from Source-Code Changes*. In Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, ASE '06, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society. 28

- [Wloka 2009] Jan Wloka, Barbara Ryder, Frank Tip and Xiaoxia Ren. *Safe-Commit analysis to Facilitate Team Software Development*. In Proceeding ICSE '09 Proceedings of the 31st International Conference on Software Engineering, pages 507–517, 2009. 61
- [Yau 1978] Stephen S. Yau, J. S. Collofello and T. MacGregor. *Ripple effect analysis of software maintenance*. In The IEEE Computer Society's Second International Computer Software and Applications Conference, pages 60–65. IEEE Press, nov 1978. 22, 67
- [Zeller 2013] Andreas Zeller. *Can We Trust Software Repositories?* In Jürgen Münch and Klaus Schmid, editeurs, *Perspectives on the Future of Software Engineering*, pages 209–215. Springer Berlin Heidelberg, 2013. 25
- [Zhang 2015] Tianyi Zhang, Myoungkyu Song, Joseph Pinedo and Miryung Kim. *Interactive Code Review for Systematic Changes*. In 37th International Conference on Software Engineering, pages 1–12, 2015. 66
- [Zimmermann 2005] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl and Andreas Zeller. *Mining Version Histories to Guide Software Changes*. *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pages 429–445, June 2005. 25, 37, 40
- [Zumkehr 2007] Pascal Zumkehr. *Changeboxes — modeling change as a first-class entity*. Master's thesis, University of Bern, February 2007. 29

A APPENDIX

A.1 PHARO Programming Language

PHARO is a SMALLTALK inspired object-oriented and dynamically-typed general-purpose language with its own programming environment. The language has a simple and expressive syntax which can be learned in a few minutes. Language concepts in PHARO are consistent, everything is an object: classes, methods, numbers, strings, even the execution context.

PHARO runs on top of a bytecode-based *virtual machine*. Development takes place in an *image* in which all objects reside. All these objects can be modified by the programmer, this includes classes and methods. Hence, we eliminate the typical edit/compile/run cycle and instead incrementally add, remove or modify classes and methods. It is worth noting that *all* classes can be extended with new methods in PHARO. For instance, one can add new operations on integers or strings, classes that are treated as unchangeable internal objects by many other high-level languages. For deployment and debugging, the state of a running image can be saved at any point and subsequently restored.

A.1.1 Minimal Syntax

Reserved Words

<code>nil</code>	the undefined object
<code>true, false</code>	boolean objects
<code>self</code>	the receiver of the current message
<code>super</code>	the receiver, in the superclass context
<code>thisContext</code>	the current invocation on the call stack

Literal Object Syntax

<code>'a string'</code>	
<code>#symbol</code>	unique string
<code>\$a</code>	the character <code>a</code>
<code>12 2r1100 16rC</code>	integers twelve in decimal, binary and hexadecimal encoding
<code>3.14 1.2e3</code>	floating-point numbers
<code>#(abc 123)</code>	literal array containing the symbol <code>#abc</code> and the number <code>123</code>
<code>#[12 16rFF]</code>	literal byte array containing the bytes/integers <code>12</code> and <code>255</code>
<code>{foo . 3 + 2}</code>	dynamic array built from 2 expressions

Reserved Characters in Expressions

<code>"a comment"</code>	
<code>.</code>	expression separator (period)
<code>;</code>	message cascade (semicolon)
<code>:=</code>	assignment
<code>^</code>	return a result from a method (caret)
<code>[:p expr]</code>	code block with a parameter
<code> foo bar </code>	declaration of two temporary variables
<code><pragma>, <primitive: 3></code>	pragma or annotations used in methods, for instances to declare a primitive method.

A.1.2 Message Sending

A method is called by sending a message to an object called the *receiver*. Each message returns an object. Messages are modeled from natural languages with a subject a verb and complements. There are three types of messages with descending precedence: unary, binary, and keyword.

Unary messages have no arguments.

```
Array new.
```

The first example creates and returns a new instance of the `Array` class, by sending the message `new` to the class `Array` that is an object.

```
 #(1 2 3) size.
```

The second message returns the size of the literal array which is `3`.

Binary messages take only one argument and are named by one or more symbol characters.

```
3 + 4.
```

The `+` message is sent to the integer object `3` with `4` as the argument.

```
'Hello', ' World'.
```

In the second case, the string `'Hello'` receives the message `,` (comma) with the string `' World'` as the argument.

Keyword messages can take one or more arguments that are inserted in the message name.

```
'Smalltalk' allButFirst: 5.
```

The first example sends the message `allButFirst:` to a string, with the argument `5`. This returns the string `'talk'`.

```
3 to: 10 by: 2.
```

The second example sends `to:by:` to `3`, with arguments `10` and `2`; this returns a collection containing `3, 5, 7, and 9`.

A.1.3 Precedence

There is a fixed global precedence when evaluating expressions in PHARO: Parentheses $>$ unary $>$ binary $>$ keyword, and finally from left to right.

```
(10 between: 1 and: 2 + 4 * 3) not
```

Here, the messages `+` and `*` are sent first, then `between:and:` is sent, and finally `not`. The rule suffers no exception: operators are just binary messages with *no notion of mathematical precedence*, so `2 + 4 * 3` reads left-to-right and thus yields `18` and not the expected `14`!

A.1.4 Cascading Messages

Multiple messages can be sent to the same receiver with `;`.

```
OrderedCollection new
  add: #abc;
  add: #def;
  add: #ghi.
```

The message `new` is sent to `OrderedCollection` which results in a new collection to which three `add:` messages are sent with different arguments. The value of the whole message cascade is the value of the last message sent (here, the symbol `#ghi`). This example is the equivalent of first assigning the new collection to a temporary variable and sending three separate `add:` messages:

```
| newCollection |
newCollection := OrderedCollection new.
newCollection add: #abc.
newCollection add: #def.
newCollection add: #ghi.
```

To return the original receiver of the message cascade (*i.e.*, the collection) instead of the last result (*i.e.*, `#ghi`), the `yourself` message is used:

```
OrderedCollection new
  add: #abc;
  add: #def;
  add: #ghi;
  yourself.
```

A.1.5 Blocks

Blocks are objects containing code that is executed on demand, (anonymous functions or closures). They are the basis for control structures like conditionals and loops.

```
2 = 2
  ifTrue: [ Error signal: 'Help' ].
```

The first example sends the message `ifTrue:` to the boolean `true` (computed from `2 = 2`) with a block as argument. Because the boolean is `true`, the block is executed and an exception is signaled.

```
#('Hello World' $!)
  do: [ :e | Transcript show: e ]
```

The next example sends the message `do:` to an array. This evaluates the block once for each element, passing it via the `e` parameter. As a result, `Hello World!` is printed.

A.1.6 Methods

Methods are first-class objects in PHARO and can be inspected and modified on the fly. Methods are created by saving expressions in the PHARO development environment. Typically methods are printed with a special first line indicating the class the method is installed on and the name or selector it is given.

```
Array >> helpMethod
  2 = 2
  ifTrue: [ Error signal: 'Help' ].
```

This example would denote a simple method with a unary selector on the `Array` class. This method could be invoked by evaluating `Array new helpMethod`.

Certain methods are marked with a pragma to use predefined primitives from the VM. These are used for expressions that cannot be expressed in PHARO. For instance the `basicNew` which allocates new objects uses the primitive number 70:

```
Behavior >> basicNew
  "Answer a new instance of this class"
  <primitive: 70>
  OutOfMemory signal.
```


Curriculum Vitae

Personal Information

Name: Martín Dias
Date of Birth: November 28, 1981
Place of Birth: Lanús, Argentina
Nationality: Argentine

Education

2012-1015: **Ph.D. in Computer Science**
RMod
INRIA, Lille - Nord Europe, France
<http://rmod.inria.fr/>

2000-2012: **Master of Science in Computer Science**
Thesis Title: *Fuel: A Fast General Purpose Object Graph Serializer*
University of Buenos Aires, Argentina
<http://www.dc.uba.ar/>

