Université des Sciences et Technologies de Lille – Lille 1

Département de formation doctorale en informatique **École doctorale SPI Lille**
UFR IEEA

Assessing and Improving Code Transformations to Support Software Evolution

THÈSE

présentée et soutenue publiquement le 28 Février 2017

pour l'obtention du

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

par

Gustavo Jansen de Souza Santos

Composition du jury

Président :	Romain ROUVOY
Rapporteur :	Serge DEMEYER, Fernando BRITO E ABREU
Examinateur :	Coen DE ROOVER, Romain ROUVOY
Directeur de thèse :	Nicolas ANQUETIL
Co-Encadrant de thèse :	Anne ETIEN

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

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

Acknowledgments

I would like to thank my thesis supervisors Nicolas Anquetil and Anne Etien for giving me the opportunity to visit the RMoD group as an intern and to ultimately do a Ph.D. in the group. I appreciate the support as educators, advisors, and fellow researchers. Also thanks to Stéphane Ducasse for welcoming me to the RMoD group, and for the valuable ideas as a heavy Pharo developer, specially in the last year of this thesis.

I thank the reviewers and jury members of this thesis: Fernando Brito e Abreu, Serge Demeyer, Coen de Roover, and Romain Rouvoy for the valuable feedback both on the thesis report and during my defense.

Many thanks to the RMoD group members. I thank Vincent Blondeau and Brice Govin for the collaboration and fellowship. I thank Usman Bhatti, Guillaume Larcheveque, Leo Perard, Marcus Denker, and Damien Pollet for the support around the Pharo world at the beginning of my stay. I thank Damien Cassou, Christophe Demarey, Anne Etien, and Jean-Christophe Bach for providing their projects for analysis. I thank Stefan Marr and Maxime Roelandt for the in-office fellowship and complicity.

I also thank Martin Dias for the support (until this day) around Epicea and other related tools. I appreciate working on such reliable tool support. Last but not least, I thank Andre Hora for welcoming me in France, helping me with boring paperwork, and introducing me to Pharo. I conclude my studies glad to have worked with this team.

Many thanks to the collaborations I was glad to have during this work. I thank the ASERG/UFMG group (Belo Horizonte, Brazil) for welcoming me during my visit, specially to Marco Túlio Valente for the collaborations, discussions, and support I thank the LASCAM/UFU group (Uberlândia, Brazil), specially Marcelo Maia and Klérisson Paixão, with whom I had the opportunity to closely collaborate. I also thank Mark Rizun for letting me collaborate in his project about pattern matching.

I thank CAPES (Brazil) for the financial support.

Finally, I thank my family and friends, including the ones I have met in France, for the support during this adventure of moving abroad.

Abstract

In software development, change is the only constant. Software systems sometimes evolve in a substantial way, for example, by migrating the system to a new architecture, or by updating the APIs on which the system depends. During this process, sequences of code transformations (*e.g.*, create a class, implement a given interface, then override a method) are systematically performed in the system (*e.g.*, to some classes in the same hierarchy or package). Due to the repetitive nature of these transformations, some automated support is needed to ensure that these sequences of transformations are consistently applied to the entire system.

One solution to deal with systematic code transformation is to allow developers to compose their own sequences of code transformations. These sequences may be defined manually, *e.g.*, by using a language to specify each transformation, or they may be identified from one or more concrete examples of transformations provided by the developer. We argue that existing approaches lack the definition of sequences that are: (i) specific to the system to which they were applied; (ii) eventually complex, *i.e.*, not fully supported by existing refactoring tools; (iii) not localized, *i.e.*, several code entities can be affected at each occurrence of the sequence; and (iv) self-aware of potential violations that might be introduced as consequence of these transformations.

In this thesis we propose to improve source code transformations to better support developers performing more complex and systematic code transformations. We cover two aspects:

- The automated support to compose and apply sequences of code transformations. We undergo an investigation on the existence of these sequences in real-world software systems. We propose a tool to automatically apply these sequences in the systems we analyzed.
- The detection of design violations during a transformation effort. We undergo an investigation on cases of systematic application of refactoring transformations. We proposed a tool that recommends additional transformations to fix design violations that are detected after performing refactoring transformations.

We evaluated the proposed approaches quantitatively and qualitatively in real-world case studies and, in some cases, with the help of experts on the systems under analysis. The results we obtained demonstrate the usefulness of our approaches.

Keywords:. software evolution, software maintenance, automated code transformation, programming by example, source code transformation

Résumé

Dans le domaine du développement logiciel, le changement est la seule constante. Les logiciels évoluent parfois de façon substantielle, par exemple en migrant vers une nouvelle architecture ou en modifiant des bibliothèques dont dépend le système. Pendant ce processus, des séquences de transformation de code (par exemple, créer une classe, implémenter une interface donnée, puis surcharger une méthode) sont systématiquement appliquées dans le systéme (*e.g.*, à certaines classes dans une même hiérarchie ou dans un même paquetage). De par la nature répétitive de ces transformations, il est nécessaire d'automatiser leur support afin d'assurer que ces séquences de transformations sont appliquées de façon consistante sur la globalité du système.

Une solution pour gérer des transformations systématiques de code est de permettre aux développeurs de composer leurs propres séquences de transformations de code. Ces séquences peuvent être définies manuellement par exemple en utilisant un langage pour spécifier chaque transformation ou elles peuvent être identifiées à partir d'un ou plusieurs exemples concrets fournis par le développeur. Nous affirmons que les approches existantes ne permettent pas de définir des séquences : (i) spécifiques au système sur lesquelles elles sont appliquées ; (ii) éventuellement complexes c'est-à-dire pas entièrement supportées par les outils de refactoring existant ; (iii) non localisées, c'est-à-dire plusieurs entités de code peuvent être impactées à chaque occurrence de la séquence ; et (iv) conscientes de possibles violations qui peuvent être introduites consécutivement à ces transformations.

Dans cette thèse, nous proposons d'améliorer les transformations de code pour mieux aider les développeurs dans l'application de transformation de code systématiques et complexes. Nous couvrons deux aspects :

- Le support automatisé pour composer et appliquer des séquences de transformations de code. Nous réalisons une recherche de l'existence de telles séquences dans de vrais logiciels. Nous proposons un outil pour appliquer automatiquement ces séquences dans les systèmes que nous avons analysés.
- La détection de violations de bons principes dans la conception lors d'efforts de transformation. Nous effectuons une évaluation sur des cas d'application systématiques de transformations de refactoring. Nous proposons un outil qui recommande des transformations additionnelles pour résoudre les violations de conception qui ont pu être détectées après avoir effectué les transformations de refactoring.

Nous évaluons les approches proposées quantitativement et qualitativement sur des cas d'étude issus du monde réel, parfois avec l'aide des experts du système analysé. Les résultats obtenus montrent la pertinence de nos approches.

Mots clés:. réingénerie logicielle, maintenance logicielle, transformation de code automatisée, programmation par l'exemple, transformation de code source

Contents

1 Introduction

Contents

1.1 Context

In software development, change is the only constant. During their lifetime, software systems must constantly evolve to remain useful in their context [Leh96]. Most of the effort during a software system lifecycle is spent to support its evolution [Som00], which is also responsible for up to 80% of the total cost of the development [Erl00].

Evolution can be achieved by code transformation activities such as correcting bugs, or adding new features to answer client requirements. These activities are frequent in the system's lifecycle [ASH13]. Their required modifications are localized, mostly inside a single method or a class. Furthermore, these activities are eventually supported by automated refactoring tools [RBJO96,Rob99], such as the one of Eclipse IDE.

It also sometimes happens that a larger and substantial effort is undertaken in the system. For example, to migrate the system to a new architecture, or to update APIs on which the system depends. Such effort is infrequent, however it consists in long periods of modifications that might affect the entire system. Moreover, such large effort is mostly done manually, *i.e.*, the required modifications are specific to the system and therefore they are not fully supported by refactoring tools. Such large effort is referred in literature as rearchitecting [ASH13].

Either performing localized or larger evolution effort, source code transformations are repetitive by nature. Sequences of source code transformations (*e.g.*, create a class, implement a given interface, then override a method) occur in the history of a system, however they are applied to distinct but similar code locations (*e.g.*, some classes in the same hierarchy). Such systematic behavior have been studied in the literature in the context of fixing bugs [NNN+13, MKM13], and adapting a system to accomodate API updates [RK12]. Specially when the entire system is modified, it is likely that sequence of transformations can be repetitive during rearchitecting efforts as well.

Additionally, software evolution is often driven by perfective and corrective maintenance, rather than improving the code quality, or system organization [OCBZ09, Pér13]. Violations of design principles, known in the literature as design smells [SSS14], might be introduced. These violations may not produce errors, therefore they might remain latent in the system until a major preventive maintenance takes place [Leh96, OCBZ09]. Additionally, there is also little support for the actual correction of design smells, once they are detected. Correction approaches often suggest how to transform the code so that the smell is removed. But concretely, transformations to remove the smell must be manually performed by the developer.

In short, software evolution is naturally a complex task. This fact may be aggravated in environments where agile software development takes place, causing negative impact on code consistency and software quality. Developers need help maintaining their systems (i) when systematic code transformations must be applied and (ii) when violations in software quality are introduced as a consequence of these transformations.

1.2 Problem

Due to the repetitive nature of some sequences of code transformations, some good practices should be adopted. For example, Integrated Development Environments (IDEs) such as Eclipse or IntelliJ should support the automation of sequences of source code transformations that can be applied in distinct code locations. Currently, IDEs provide refactoring, *i.e.*, a sequence of code transformations limited to a subset of behavior-preserving transformations.

However, recent work sparked discussion about the lack of trust in refactoring tools [AGF11, NCV+13, VCM+13]. Developers do not understand what most of the refactoring transformations actually do, and sometimes they prefer to perform code transformations manually, even though there is an automatic refactoring that produces the same outcome. Additionally, refactoring relies on simple and localized transformations. Large transformation efforts such as rearchitecting are therefore not fully supported, and developers must perform the transformations manually.

Many problems may occur due to the repetitiveness of the transformations and to the fact that these transformations are applied manually. Below we list some concrete problems detected in the literature:

- The application of sequences of code transformations in similar code locations is a tedious task. Developers have to identify all the code locations that are candidates for transformation, then manually apply the sequence of transformations in each code location [VEdM06, AL08, MKM13, JPW+15].

- The application of code transformations in each code location is a complex task. Different code locations, *e.g.*, methods, might introduce variables with different names and different argument values, for example. Developers have to identify which variables and arguments are the correct candidates for transformation [CDMS02, KN09, MKM11].
- The manual application of sequences of code transformations is an error-prone task. First, developers might not know, from all the code entities in a system, which of them are candidate for transformation; therefore introducing errors of omission. And second, given a new code location, developers might forget to apply some of the transformations in the same sequence as they were originally performed [AL08, RL08, NNP^{+}10, MKM13].
- After transforming all candidate locations, developers have to also check whether violations of design principles were introduced as a consequence of this repetitive transformation. The occurrence of smells, such as God Class and Shotgun Surgery, is likely to increase the occurrence of bugs [LS07] and to reduce quality factors such as understandability and maintainability [DSA^{+}04,SSS14]. Moreover, smells might remain unnoticed in the system after they are introduced [CM14].

These problems show the difficulty of software evolution, in particular to keep consistency of source code and its quality after a large evolution effort. It is important to ensure that source code evolution is correctly applied.

1.3 Supporting Systematic Code Transformation

To facilitate the time-consuming task of systematically transform similar code, approaches were proposed to support the definition of sequences of code transformations for at least three decades [Nei84, ABFP86]. These approaches allow the developer to define their own sequences of transformations; therefore our discussion does not include refactoring approaches, such as Eclipse, which have a predefined set of transformations that are applied separately.

France *et al.* [FGSK03] propose to transform models by applying design patterns. For this purpose, they specify (i) the problem corresponding to the design pattern application condition, (ii) the solution corresponding to the result of the pattern application and (iii) the transformation corresponding to the sequence of "*operator templates*" that must be followed in order for the source model to become the target model. Similar definition approaches based on condition and operators are also proposed by Markovic and Baar [MB08], and Lano and Rahimi [LR13]. Defining composite transformations for the application of design patterns is *generic* in the sense that transformations can be applied to systems of different domains and even different programming languages.

Other approaches work with existing source code. Similar to most model approaches, Kozaczynski *et al.* [KNE92] also propose composite transformations using application conditions and operators. There are also approaches which are based on code examples [CDMS02, BPM04]. The transformation is defined from an example of the source code before and after the transformation. However, in these approaches, the operators are *simple* and consist in short sequences of code insertion, replacement, and/or deletion.

Coccinelle relies on text matching to define bug patches [LLH+10]. A composite transformation is defined as a set of variable declarations, followed by a list of code deletions and insertions. Variables represent code entities such as expressions and statements. However, the matching and transformation process is restricted to in-file operators. Concerning the complexity of the operators involved, most code transformation approaches are *localized* and modify only one entity at each time (*e.g.*, the code inside a method or a file).

Specially when code transformations are being applied to the system at large, it is expected that the source code undergoes some intermediate and unstable state until the system is completely transformed. In this context, code quality violations might be introduced and remain in the system until an explicit preventive effort takes place [Leh96, OCBZ09].

In summary, existing approaches lack the definition of sequences of code transformations that are (i) system specific, *i.e.*, they cannot be applied to other systems; (ii) eventually complex, in terms of the number of operators involved; (iii) not localized, meaning the sequence of transformations might affect several code entities instead of a single method; and (iv) self-aware of potential quality violations that the transformations themselves might introduce.

1.4 Our Approach in a Nutshell

In this thesis we propose to improve source code transformations to better support developers specially, but not exclusively, during a rearchitecting effort. We cover two aspects: (i) the automation of sequences of transformations composed by the developer that can be applied to the system at large, and (ii) the detection and automatic correction of violations that might be introduced when a code transformation is performed.

Supporting Systematic Code Transformation. Existing approaches that support automated code transformation mostly rely on refactoring transformations that can be applied to systems of different domains. Sequences of transformations that might affect the system at large are not easily supported. We first undergo an investigation on the existence of sequences of code transformations that are specific to the systems they were applied. Then, we propose a tool to compose and automatically apply these sequences in the systems we analyzed.

Supporting Detection of Quality Violations. Violations of design might be introduced when a system undergoes systematic transformations. We propose an approach to detect design smells and suggest their correction right after the developer performs a refactoring transformation. We undergo case studies with real cases of systematic code transformation to check whether the suggested transformations are accepted by developers.

1.5 Contributions

The main contributions of this thesis can be summarized as follows:

- **Contribution 1**. We demonstrate the existence of repetitive sequences of code transformations during a large evolution effort in real software systems. We validate the properties that make these sequences of transformations particularly challenging to apply manually [SAE+15c].
- **Contribution 2**. We provide automated support to record and replay sequences of code transformations. These sequences can be configured, manually or automatically, to be later applied in several code locations [SAE+15b].
- **Contribution 3**. We provide automated support for detection and correction of quality violations during real and large evolution efforts. Our solution focuses on the detection and automatic correction of design smells after a code transformation is performed by the developer.

1.6 Structure of the Thesis

Chapter 2: Motivation

This chapter presents our first study on a rearchitecting case. This study serves as a motivation for this thesis, whereas we discovered a case of systematic code transformation that would benefit from an automated support.

Chapter 3: State of the Art

This chapter presents the related work in the context of identifying systematic code transformation, providing support to automatically transform code, and identifying quality violations, in particular, design smells.

Chapter 4: Relevance of Systematic Code Transformation

This chapter describes an investigative study to analyze the benefits of automating sequences of code transformations that are defined by the developer. We undergo an investigation with real cases of rearchitecting to understand whether the transformations involved are worth being automated in the context of a large transformation effort.

Chapter 5: Supporting Systematic Code Transformation

This chapter describes our first approach to automatically perform sequences of system-specific code transformations. We record code transformations from the development tool, then we configure these transformations to replay them at other locations in the system. We validate our approach on the cases of systematic code transformation we discovered in Chapter 4.

Chapter 6: Automating Systematic Code Transformation

This chapter describes two approaches to automate the process of replaying sequences of code transformations. The first approach focuses on reducing the effort of configuring these sequences to replay in another location, which was originally manual. The second approach focuses on recommending locations in source code that are candidate for systematic code transformation, after one sequence of transformations is recorded. We validate these approaches on the same systems we analyzed in Chapters 4 and 5.

Chapter 7: Improving Code Transformations

This chapter presents our analysis of the impact of refactoring transformations in the introduction of design smells. We cover the Pharo[1] ecosystem, which has six years of evolution, about 3,600 distinct systems, and more than 2,800 contributors, where two refactoring transformations were systematically performed, *e.g.*, *Move Class* and *Extract Method*.

Chapter 8: Conclusion

This chapter concludes the thesis and presents future work.

[1] `http://pharo.org/`

2 Motivation

Contents

In this chapter, we present our first work to support software evolution. This work was part of a larger project, in collaboration with ASERG Group in Brazil, and it specifically focused on supporting architecture evolution. However, it is worth noting that this study is a motivation for our work. Our thesis focuses on code transformations instead of architecture evolution. Nevertheless, the results of this study served as guidelines to focus our thesis onto supporting systematic code transformation.

2.1 Introduction

The definition of the right software architecture in early stages of software development is of utmost importance as a communication mechanism between stakeholders. In this thesis, we consider Garlan and Perry's definition of architecture: the structure of components of a program, their relationships, and principles and guidelines governing their design and evolution over time [GAO95]. Architecture definition does not only involve the decisions and guidelines that must be followed during the software evolution, but also provides discussion about the requirements (and conflicting parts of them) that the software might have.

However, software systems are under continuous evolution [DDN02, Leh80]. Their architecture inevitably gets more complex, harder to understand, and further modifications become more difficult to implement. In addition to an increase in complexity, it is common to observe modifications that do not follow the architecture as originally proposed for the system [DP09,PTV$^+$10]. Increasing complexity and architectural violations tend to be neglected over the years, unless explicit effort is done to improve the architecture [Leh96, SRK$^+$09].

In this chapter, we focus on the definition of an architecture description, with the goal to disseminate and document architectural knowledge among developers. Additionally, we also focus on architecture conformance, which goal is to verify whether the implementation is consistent with the intended architecture.

The main contributions of this chapter are:

- We present a prototype tool, called OrionPlanning [SAE$^+$15a]. Using this tool, the user can create an architecture definition from scratch or iteratively modify an existing one extracted from source code (Section 2.3).
- We evaluate OrionPlanning in a simple system that illustrated a real case of architectural deterioration. Architectural rules were defined by the user, *i.e.*, they were specific for the system under analysis. Our tool was able to detect architectural violations during the evolution of this system (Section 2.4).
- We discovered work that is later developed in this thesis in two directions: (i) the automated support for systematic code transformation (Chapter 5), and (ii) the continuous detection of violations when the software is under evolution (Chapter 7).

Structure of the Chapter

Section 2.2 presents related work on architecture definition tools. Section 2.3 presents the OrionPlanning tool. Section 2.4 presents the evaluation of the tool in a case of software evolution, and Section 2.5 concludes the chapter.

2.2 Related Work

Restructuring the architecture is usually required to keep complexity under control and to repair eventual architectural violations. Such task typically requires a sequence of transformations applied over different components, optionally followed by continuous evaluation of the architectural gains achieved by these transformations. Restructuring is then more complex than isolated refactoring because the entire system is involved. Additionally, it is also a challenging task because it is usually performed in an *ad-hoc* way [TVCB12].

Many approaches to define and evolve software architecture have been proposed in the literature. However, there is still little adoption of existing approaches in industry. Previous work analyzed this gap between research and practitioners' needs [Cle96, HR98, HA11]. Recently, Malavolta *et al.* [MLM$^+$13] conducted a survey with real practitioners concerning architectural languages *i.e.*, any form of expression, informal or formal, to describe software architecture. The authors identified features which practitioners considered useful in past projects, and we list the most important ones as follows.

- *Iterative Architecting:* the ability to refine the architecture from a general description. It means that the language should not require the stakeholders to fully describe the architecture at the beginning.
- *Analysis:* the ability to extract and analyze information from the architecture for testing, simulation, and consistency checking, for example. One of

the clearest result from the study was the need for proper analysis to detect inconsistencies in the architecture before it would be applied.

The survey conducted by Malavolta *et al.* [MLM^{+}13] cites architectural languages such as UML and AADL. In this section, we select architecture description approaches that apply to two main conditions: (i) the approach must provide a tool to support architecture description, and (ii) the approach must also provide analysis, *e.g.*, dependency checking, on the proposed architecture.

That *et al.* [TSO12] use a model-based approach to document architectural decisions as architectural patterns. An architectural pattern defines architectural entities, properties of these entities, and rules that these properties must conform to. The approach provides analysis by checking the conformance between an existing architecture definition and a set of user-defined architectural patterns.

Baroni *et al.* [BMMW14] also use a model-based approach and extend it to provide semantic information. With assistance of a wiki environment, additional information is automatically synchronized and integrated with the working model. The analysis consists in checking which architectural entities are specified in the wiki. One critical point of this approach is that the information might be scattered in different documents, which can be difficult to maintain.

2.3 OrionPlanning in Action

Figure 2.1 depicts the main user interface of OrionPlanning. It is build on top of the Moose platform [BAD12]. The panel in Figure 2.1.A shows the system under analysis and its versions, followed by a panel for color captions (Figure 2.1.B), and the list of model changes in the selected version (Figure 2.1.C). On the right side of the window, OrionPlanning generates a simple visualization of model entities and dependencies (Figure 2.1.D) and a list of dependency constraints which will be evaluated when the model changes (Figure 2.1.E).

2.3.1 Loading Code

To modify an existing project with OrionPlanning, the Moose platform already provides support to import code written in C++, Java, Smalltalk, Ada, Cobol, and other languages. The result is an instance of the FAMIX meta-model [DAB^{+}11]. FAMIX is a family of meta-models that represents source code entities and relationships of multiple languages in a uniform way. We chose FAMIX because Moose already provides inspection and analysis tools which can be extensible for our work (see Section 2.3.4). Details on how to import models in Moose are provided in The Moose Book [Moo10]. After importing the code, a new model appears in the model selection panel (Figure 2.1.A).

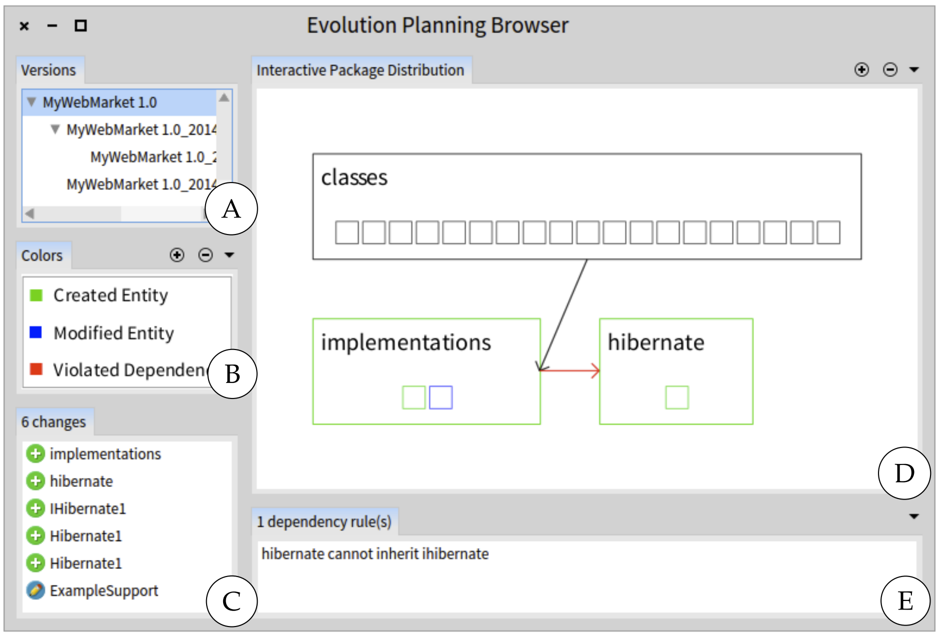


Figure 2.1: OrionPlanning overview. The panel (D) shows three packages.

2.3.2 Versioning

We use Orion [LDDF11] to perform transformations on the FAMIX model extracted in the previous step. Orion is a reengineering tool that simulates transformations in multiple versions of the same source code model. Orion efficiently handles the creation of *children* models. A *child* model has one reference to its parent version and a list of changes that were made in it. Orion manages modifications in multiple versions, including merging and resolving conflicts, without creating copies of the source code model.

Figure 2.1.A shows the panel for model management. In practice, from a given model, the user can (i) inspect the changes in the current version, *i.e.*, check the list of changes and modified entities; (ii) eventually create a new child version and make changes in it; and/or (iii) discard the current version for different reasons. When the user is modifying a child model, the original one is not modified and no copies of this model are created. The list of changes is also displayed in OrionPlanning's main window (see Figure 2.1.C).

2.3.3 Restructuring and Visualizing

In order to provide architecture visualization, we use a visualization engine called Telescope. Figure 2.2 illustrates a visualization of a Java project. Packages are rectangles with classes represented as squares inside the package. Both packages and classes are expandable by mouse click. After expansion, a class shows its methods as small squares. In general, entities that were changed in the current model have their borders colored in blue, and the borders of entities created in the current model are colored in green.

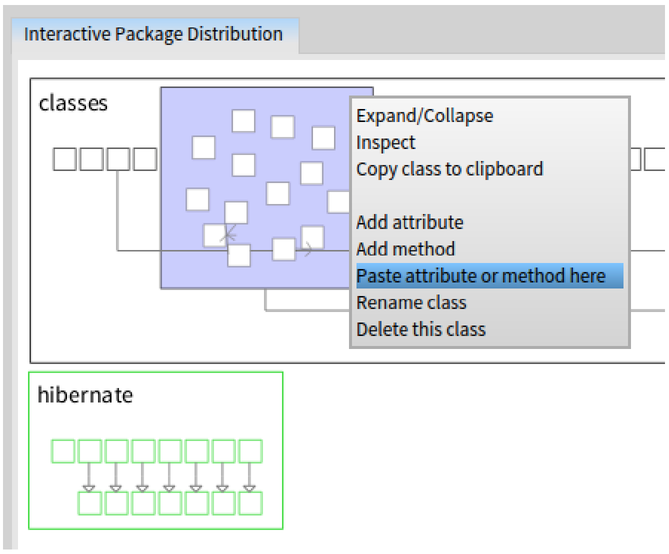


Figure 2.2: OrionPlanning's interactive visualization (right click on a class).

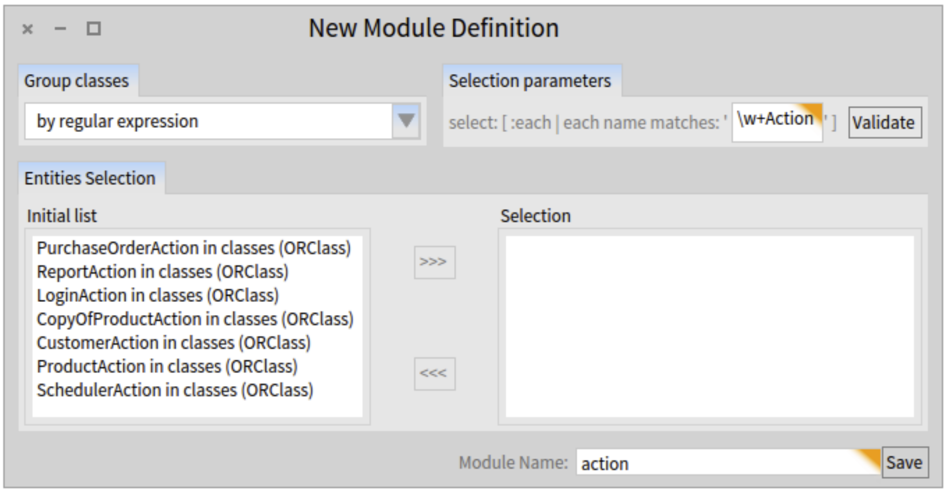
The visualization displays three types of dependencies: (i) the *package* dependency, represented as arrows in Figure 2.1.D, summarizes all dependencies (*i.e.*, accesses, references, invocations, etc.) at a package level; (ii) the *class* dependency, represented as empty arrows in Figure 2.2 (classes in hibernate package), shows inheritance dependencies between classes inside one package; and (iii) the *method* dependency, represented as small arrows inside highlighted class in Figure 2.2, shows invocations between methods of the same or different classes. We decided to show a fraction of all the dependencies to not overload the visualization with edges. Refactoring-based operators (*e.g.*, add, paste attribute, and remove entities) are accessible by right click menu, as shown in Figure 2.2.

2.3.4 Model Analysis

According to previous survey with practitioners, the feature they missed the most in past projects was the support for architectural analyses [MLM$^+$13]. Some of suggested analyses include dependency analysis between entities in an architecture, and consistency of architectural constraints [PW92]. In this section, we describe our work on extending OrionPlanning to provide dependency checking constraints defined by the user.

Moose provides a set of metrics for software, such as size, cohesion, coupling, and complexity metrics. From the visualization provided by OrionPlanning, any entity can be inspected and evaluated at any time (right click, Inspect). OrionPlanning allows the user to define rules based on these metrics. A possible example consists in restricting the number of classes in a package to less than 20.

OrionPlanning also supports the definition of dependency constraints. The definition uses the same syntax of DCL [TVCB12], a DSL and tool for conformance checking originally proposed to Java systems. In OrionPlanning, the user first defined logical *modules* as a set of classes. These classes can be selected by matching a property (*e.g.*, a regular expression), or by manually selecting them into the module. Figure 2.3 depicts the module definition browser, in which the user selected (by regular expression) all classes which name ends with "*Action*". Other properties can be easily extended to the model.



Group classes: by regular expression

select: [:each | each name matches: '\w+Action']

PurchaseOrderAction in classes (ORClass)
ReportAction in classes (ORClass)
LoginAction in classes (ORClass)
CopyOfProductAction in classes (ORClass)
CustomerAction in classes (ORClass)
ProductAction in classes (ORClass)
SchedulerAction in classes (ORClass)

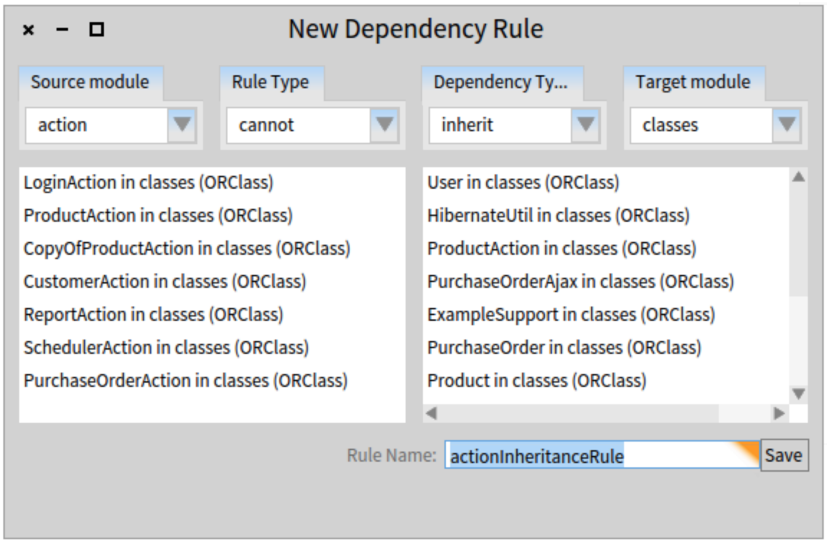
Module Name: action

Figure 2.3: OrionPlanning's model definition browser

Finally, the user defines a dependency constraint between two logical modules, which are defined in the previous step. Given two modules A and B, DCL defines the following constraints:

- only A can depend on B;
- A can only depend on B;
- A cannot depend on B; or
- A must depend on B.

Dependencies include access to variables, references to class, invocation to methods, and inheritance to classes. Figure 2.4 shows the rule definition panel, in which the user selects the source module, the type of constraint, the type of dependency to be analyzed, and the target module. In this example, the user defined that all Action classes cannot inherit from classes in the original monolithic package (named classes).


New Dependency Rule

Source module	Rule Type	Dependency Ty...	Target module
action	cannot	inherit	classes

LoginAction in classes (ORClass)
ProductAction in classes (ORClass)
CopyOfProductAction in classes (ORClass)
CustomerAction in classes (ORClass)
ReportAction in classes (ORClass)
SchedulerAction in classes (ORClass)
PurchaseOrderAction in classes (ORClass)

User in classes (ORClass)
HibernateUtil in classes (ORClass)
ProductAction in classes (ORClass)
PurchaseOrderAjax in classes (ORClass)
ExampleSupport in classes (ORClass)
PurchaseOrder in classes (ORClass)
Product in classes (ORClass)

Rule Name: actionInheritanceRule

Figure 2.4: OrionPlanning's dependency constraint browser.

In order to check the conformance between a model and the defined constraints, OrionPlanning queries all the dependencies in the current model. The dependency checker analyzes each dependency with each constraint. Violations to the constraints are highlighted in the visualization with a red color (see Section 2.4 in our concrete example). Finally, the dependency checker listens to changes in the current model and checks all the dependencies when a change is performed.

2.4 OrionPlanning Evaluation

To illustrate the use of OrionPlanning, we use a simple e-commerce system, called MyWebMarket. This system was created independently by the ASERG group to illustrate a case of architectural erosion and the analysis of architectural violations [TVCB12]. This system was developed in a sequence of versions. The first version follows a very naive implementation and successive versions continuously improved the modularization to correct specific dependency constraints.

In our evaluation, we imported the first version of MyWebMarket, consisting of only one package and performed in OrionPlanning the transformations that had been made on the actual system. We discovered a first case of systematic code transformations, originally performed on code by MyWebMarket's developers, without any form of automated support. Pattern 2.1 presents an informal description of these transformations. The goal was to isolate the dependencies to a framework (*e.g.*, HIBERNATE) into a new package. Furthermore, it was decided to use the Factory design pattern.

PATTERN 2.1: Systematic transformations in MYWEBMARKET's restructuring (A package *PHib* was created to hold dependencies to Hibernate, a factory class *FHib* was created in *PHib*)

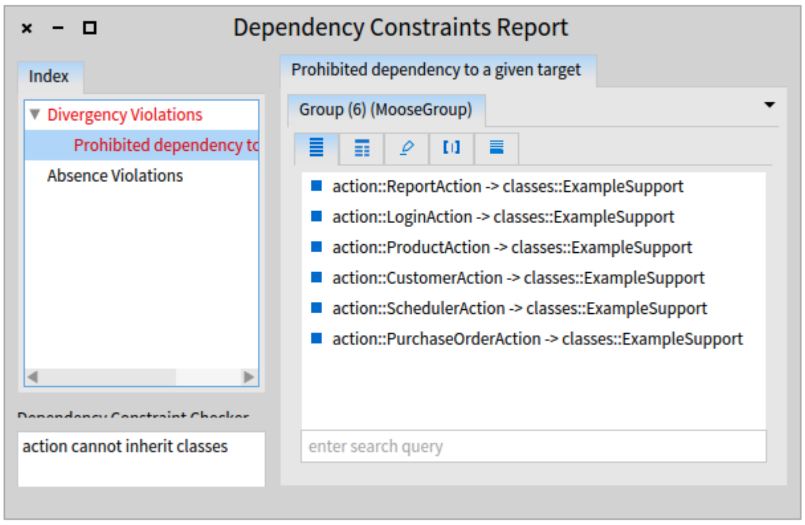
```
For each class $C \notin$ package $PHib$ that depends on Hibernate
1. add interface $IC'$ in $PHib$
2. add class $C'$ in $PHib$ implementing $IC'$
3. add method "public C' getC'()" in the factory $FHib$
   $\exists$ method $M$ in $C$
   and $\exists$ $S$ statements $\in M$ creating the dependence on Hibernate
4.    add method $M'$ in $C'$ containing statements $S$
5.    replace statements $S$ by a call FHib.getC'().M'()
```

Definition 1 A *code transformation* is a general term that refers to code addition, removal, or modification.

In Pattern 2.1, five code transformations were performed: one *Add Interface*, one *Add Class*, one *Add Method* in the factory class, then a subsequence of one *Add Method* in the new class, and one (or more) *Replace Statement* for each statement depending on the framework. Note that the two last transformations are equivalent to a *Extract Method* transformation. In this definition, we focus on the more elementary transformations; we discuss the complexity of code transformations in Chapter 3.

In addition to replaying the transformations in the model, we defined a dependency constraint on the model under analysis. We performed the constraint definition discussed in Section 2.3.4, *i.e.*, prohibit Action classes to inherit non-Action classes. In order to simplify the visualization, we previously moved all the Action classes to a new package, named action.

Figure 2.5 shows the dependency analysis in ORIONPLANNING. This view shows the list of constraint violations in the version under analysis. In this case, all of the Action classes extend a common class, named ExampleSupport, which does not follow the name convention. In order to fix this violation, the user shall move ExampleSupport to the action package, and optionally change the class name to the *Action name convention.


Dependency Constraints Report

Index

Divergency Violations
Prohibited dependency to
Absence Violations

Dependency Constraint Checker

action cannot inherit classes

Prohibited dependency to a given target

Group (6) (MooseGroup)

- action::ReportAction -> classes::ExampleSupport
- action::LoginAction -> classes::ExampleSupport
- action::ProductAction -> classes::ExampleSupport
- action::CustomerAction -> classes::ExampleSupport
- action::SchedulerAction -> classes::ExampleSupport
- action::PurchaseOrderAction -> classes::ExampleSupport

enter search query

Figure 2.5: OrionPlanning's dependency rules visualization.

2.5 Conclusion

We presented OrionPlanning, a prototype tool to support architecture evolution. In practice, the user can (i) generate a new architecture definition from scratch, or incrementally modify an existing one using refactoring-based transformations; and (ii) actively check user-defined architecture rules in the working architecture.

Most importantly, when replicating the transformations, we observed instances of systematic transformation. More specifically, we observed sequences of transformations (*e.g.*, create an interface, then create a class implementing this interface, add methods to this class) that were repetitively applied to several locations, *e.g.*, all classes depending on Hibernate. We list below two lessons learned from this replication study.

- With OrionPlanning, we were able to manually perform these sequences of transformations. However, we observed that the sequences were essentially very similar and, therefore, they would benefit from an automated support.
- When applying these sequences of transformations, architectural violations were introduced in the model, and automatically detected by our tool. Additional transformations, *e.g.*, *Move Class*, were recommended to correct these violations. This correction would also benefit from an automated support.

These results serve as guidelines to focus our efforts on (i) providing support for the developer to define and automatically apply their own sequences of transformations; and (ii) providing support for automatic detection and correction of violations when systematic code transformations are being applied.

3 State of the Art

Contents

3.1 Introduction

In this chapter we present the current approaches to support systematic code transformation, and we discuss studies on the evolution of quality violations, in particular, design smells. On the subject of systematic code transformation, we show that most of the approaches do not support large transformation efforts such as the ones we work in this thesis. First, the transformations are generic, in the sense that they can be applied in systems of different domains. Second, the transformations are simple, *e.g.*, they are composed of a short sequence of additions and removals. And third, the transformations are localized and limited in their scope, *e.g.*, they are generally small and mostly applied to one method at the time.

Structure of the Chapter

Section 3.2 presents approaches that identify repetitive sequences of code transformations from the code history of a system. Section 3.3 presents approaches for defining sequences of code transformations from small and existing ones. Section 3.4 presents approaches to automate the application of these sequences. Section 3.5 presents previous studies on the evolution and negative impact of design smells. Section 3.6 presents approaches that recommend additional code transformations after the developer transforms code, and Section 3.7 concludes this chapter.

3.2 Identification of Systematic Code Transformations

Developers and researchers alike have long perceived the existence of repetitive sequences of code transformations. More recently, Mining Source Repositories

(MSR) approaches obtained more highlight in the state of the art. MSR approaches allowed researchers, for example, to find evidences of systematic code transformations in large code repositories and over several software systems. In this section, we discuss related work on mining source code repositories to identify patterns of code transformations. It is not the scope of this thesis to propose an approach to identify such systematic behavior, but to assist the developer in defining and applying known sequences of transformations. Nevertheless, we discuss such approaches to justify their limitations in practice.

Concerning the context in which sequences of transformations are discovered, Ying *et al.* [YMNCC04] looks for groups of files that were frequently modified together, to recommend candidate files to transform based on similar transformations in the past. Fluri *et al.* [FGG08] identifies patterns of transformations, *e.g.*, change return type then rename method, which describe specialized development activities. Some approaches focused on precise activities, such as identifying patterns of bug fixes [LZ05b, PKJ09] and API updates [KNGWJ13]. Other approaches identified system-specific sequences of transformations [KBN07, NNN$^+$13]. The identified sequences are not fully supported by existing refactoring tools [FGG08, NNN$^+$13, NCDJ14], and they might be performed in not one but several versions of the system [JPW$^+$15]. Additionally, mining approaches rely on the fact that sequences of transformations were already applied in the past. Therefore, new occurrences of these sequences of transformations cannot be automatically performed in the system. We address the automation of sequences of code transformations in Section 3.4.

Concerning how code transformations are extracted from code repositories, most approaches represent code transformations as addition and removal of nodes in the AST [FGG08, NNN$^+$13, NCDJ14, JPW$^+$15]. Optionally, some composed but still simple transformations, such as renaming and updating, are also included [KZJZ07]. Additionally, some work calculate transformations that are limited in scope. For example, Kim *et al.* [KNGWJ13] consider changes until the level of the method header, *i.e.*, arguments and return types, therefore excluding transformations inside the method body. Livshits and Zimmermann [LZ05b] only consider the addition of method calls, and Dagenais and Robillard [DR08] only consider method removal. Other non-AST approaches calculate files that changed together [YMNCC04] and replacement of code fragments as a text [AL08].

Finally, concerning the algorithm to mine code transformations, most approaches use association rule mining [YMNCC04, KNGWJ13, JPW$^+$15]. The goal of the association rule algorithm consists in finding groups of similar transactions (*i.e.*, transformations) that occur with similar properties (*e.g.*, methods with the same name). Other approaches rely on frequent pattern mining such as Apriori [LZ05b], agglomerative hierarchical clustering [FGG08], and largest common subsequence [AL08].

Summary. Approaches that identify systematic code transformation do not provide automation for the sequences of transformations they identify. Most of these approaches retrieve sequences that are: (i) not fully supported by existing refactoring tools, (ii) simple, *e.g.*, a short sequence of additions and removals inside a method, and (iii) limited in their scope, *i.e.*, they do not cover all possible code transformations. In contrast, the study of system-specific and more complex sequences of transformations is not yet covered. In Chapter 5 we identify sequences of code transformations that are specific to the system they were found, and which systematically modified several methods of several classes in the same hierarchy.

3.3 Definition of Sequences of Code Transformations

Examples of systematic code transformation might be discovered in early stages of a large transformation effort. For example, developers might notice, after a few times, that they are applying the same sequence of transformations in the classes of same package or hierarchy. Additionally, several other classes shall also be transformed in a similar way. Existing approaches provide support for the definition of these sequences for later automation, to reduce the tendency for mistakes due to applying them manually, and to ease the work of developers. We specifically address the automation of these sequences in Section 3.4.

The definition of composite code transformations have been proposed in the literature for at least three decades [ABFP86, EKN91]. These work already observed that some small transformations are commonly reused by composite ones. For example, *Generate accessor* method for an instance variable and the *Extract Method* refactoring transformation reuse the same transformation: *Add Method*. Concerning the complexity and the domain where these transformations may be applied, we follow the categorization in three levels introduced by Javed *et al.* [JAP12].

Level one transformations are atomic and describe generic elementary tasks. They are atomic because they describe the addition or deletion of *one* code entity. For example, these transformations are routinely proposed in development helpers (*e.g.*, *Add Method*), or calculated from source code examples in the ChangeDistiller tool [FWPG07]. These transformations are generic in the sense that they are independent of the system, the application domain, and sometimes even the programming language.

Level two transformations are aggregations of level one transformations. For example, the *Extract Method* is a composition of several atomic transformations: *Add Method*, then a sequence of *Remove Statement* from the former method and corresponding *Add Statement* in the new method. Level two transformations describe more specific tasks than level one transformations.

However, they are still generic because they can be applied to systems from different domains.

Level three transformations are aggregations of level one or level two transformations. They are specific to the system to which they are applied. The example in Chapter 2 is specific to MyWebMarket system.

Concerning the representation of the transformations, some approaches rely on a Domain Specific Language (DSL) [VEdM06, LT12, LR13, KBD15]. Specially in the context of a large transformation effort, which might be occasional, these approaches increase the complexity of the automation process (*i.e.*, to learn another language), and they might discourage their use by the developers. Other approaches represent a composite transformation as a sequence of text insertions and deletions [CDMS02,BPM04,LLH+10,RI14]. Text-based approaches might not provide support when applied into a slightly different new code location. For example, in two methods, the names of their arguments and temporary variables might not be the same. The set of basic transformations to compose is limited to simple transformations, *e.g.*, only adding or removing a line.

Finally, concerning the context in which sequences of transformations are applied, these approaches were used in the past to apply design patterns [LR13, KBD15], to apply bug patches in similar code locations [LLH+10], to update the API on which a client system depends [HD05, NNP+10], or to apply a limited set of behavior-preserving transformations, *i.e.*, refactoring [VEdM06, MB08, LT12].

Summary. Most approaches provide a set of basic (level one) transformations, *e.g.*, add or remove a line of code. Moreover, the activities in which definition is supported, *e.g.*, fix a bug or update an API, require small sequences of transformations, *i.e.*, they modify a few code entities at each time. In Chapter 5 we propose to define a sequence of transformations by recording one example of systematic transformation from the developer. We discuss other example-based approaches in Section 3.4 because they also provide automation of the sequences of transformations that are defined.

3.4 Automation of Sequences of Code Transformations

Automation of code transformations have been proposed in the state of the practice in the form of refactorings. Integrated Development Environments, such as Eclipse, include refactoring transformations as a way to automate composite transformations that define behavior-preserving tasks. The first tools date back to almost two decades ago [RBJ97, BR98]. Most of the automated transformations are inspired by the refactoring catalog proposed by Fowler *et al.* [FBB+99].

Recent work however proved that refactoring tools are underused. Murphy-Hill *et al.* [MHPB09] and Negara *et al.* [NCV$^+$13] conducted different studies based on the refactoring tools proposed by Eclipse platform. Both studies lead to the conclusion that, when a refactoring transformation is available for automated application, developers prefer to perform the transformations manually. Vakilian *et al.* [VCM$^+$13] found similar results based on both a survey and a controlled study with professional developers. Developers do not understand what most of operators proposed by refactoring tools do, or they do not perceive how the source code will actually change after their application. Therefore, developers prefer to perform a sequence of small well-known refactoring transformations that will produce the same outcome as a composite, sometimes complex, built-in refactoring transformation.

This state of the art shows that developers need automated support to perform systematic transformations. As discussed in Section 3.2, the transformations involved might not be fully supported by existing refactoring tools. Moreover, the task of performing systematic transformations requires the modification of similar, but non-identical code locations, *i.e.*, names of classes, methods, variables, might not be the same from one occurrence to the other. This section presents related work on automating code transformations. We discuss related work in two directions: (i) automatically performing sequences of code transformations in different code locations (Section 3.4.1); and (ii) automatically identifying all code locations that are candidate for systematic transformation (Section 3.4.2).

3.4.1 Performing Systematic Code Transformations

Programming by Demonstration (or Programming by Example) is a term mostly used in robotics to comprise approaches that register and automate sequences of operations one wants to perform [Lie01, RL08]. According to this paradigm, the user provides concrete examples of how to perform the operations. As concrete examples, the feature *Search and Replace* in most text editors, the multiple selections feature in SublimeText, and the macro feature in Microsoft Office and Emacs are examples of programming by demonstration approaches. In this section, we focus on programming by demonstration approaches that propose the automation of source code transformations.

ChangeFactory [RL08] is one of the first tools to apply programming by demonstration for code transformation. In practice, the tool records one sequence of code transformations manually performed by the developer, *e.g.*, adding a method in a class, then adding statements to this method. Each code entity transformed by the developer is a parameter of the transformation. Developers can edit the parameters, *e.g.*, the name of the method to be added derives from the name of the class. Finally, the developer can test and apply the composed transformation on another code location. In Chapter 5, we present an automated code transformation approach that is inspired by ChangeFactory. We discuss our contributions in that chapter.

CatchUp! [HD05] is a similar record-and-replay tool that relies on a limited set of refactoring transformations recorded in the Eclipse IDE. Replaying the recorded refactorings also requires manual configuration by the developer. Other approaches were inspired by the record-and-replay approach [OM08, HDLL11, MKOH12]. However, they do not propose the configuration of transformations to apply them in other locations. Their goal was to understand and replay past transformations from a source code repository.

Some approaches rely on code examples that specify the result of transformations. A code example basically expresses how the source code looks *before* and *after* a sequence of transformations. These approaches either infer the transformations that took place, or replace excerpts of the example according to a new code location. We now discuss three tools which rely on code examples.

Sydit [MKM11] relies on one code example. The tool generates an *edit script* in terms of added, removed, and modified AST nodes. The configuration is done automatically by computing dependencies inside the code example. For example, modified AST nodes (between the code before and after) are considered as parameters of the transformation. Unmodified code is then used to automatically configure these parameters.

The Ekeko/X tool [RI14,MR14] also relies on one code example. In practice, the developer incrementally generalizes the example by introducing meta-variables and wildcards. During this stage, the developer can check the result of this configuration, *e.g.*, the collection of code entities that are candidate to be transformed. Similarly to this tool, Critics [ZSPK15] relies on manual configuration by the developer. However, the generalization is limited to types of classes, variable names, and method names.

LASE [MKM13] is an extension of Sydit. The tool relies on two or more code examples, and it generalizes the edit script according to similarities in these examples. For example, if the same transformation is applied to two different code entities, the approach generalizes entities as meta-variables. The accuracy of the resulting edit script depends on the given examples. If two or more code examples are too similar, the tool will not be able to replay the transformations in slightly different code locations (over specification). On the other hand, if two or mode code examples are too dissimilar, the tool might be able to apply the transformations in undesired locations (over generalization).

Summary. Most of the approaches represent a sequence of code transformations as excepts of code, *i.e.*, as an example of code before and after the transformations. In fact, most of these approaches were applied to perform small sequences of code transformations that are most likely to be modify one method at each time. In this thesis, we take inspiration from the ChangeFactory tool [RL08], *i.e.*, code transformations are recorded from developer editions in a development tool. We present our approach in Chapter 5.

3.4.2 Recommending Code Locations to Perform Transformations

Once developers defined the sequence of code transformations that must be performed systematically, they might not know, from all the code entities in the system, which ones are candidate for transformation. In this section, we discuss approaches that recommend candidate locations for automatic transformation.

Some approaches proposed to find opportunities to apply known refactoring transformations. For example, Khomh *et al.* propose the detection of God classes to recommend the application of Extract Class refactoring [KVGS09]. Bavota *et al.* [BDLMO14] discuss state of the art approaches that recommend the application of other refactorings described in Fowler's catalog [FBB+99]. Refactoring transformations such as Extract Class have well defined purposes, therefore these approaches search for very specific properties in source code for recommendation, *e.g.*, identify classes with too many methods [JJ09, HKI08, Mar04, SSL01]. In our work, the transformations we found are specific to the system on which the developer is working. The rationale behind the transformations, *i.e.*, the transformations involved and the code entities that are transformed, are different for each system.

Concerning other recurring transformations, LibSync [NNW+10] and APIEvolutionMiner [HEA+14] focused on updating the API on which a system depends. These tools extract code transformation rules from other systems that updated to the same API usage in the past, then they recommend locations in source code and transformations to replace old API calls to new ones. FixWizard [NNP+10] focused on recurring bug fixes. In the code history of five real open-source projects, up to 45% of bug fixing transformations were repetitive. Based on the recurring examples the authors found, the tool also recommends both code locations and required transformations to fix the bug.

PR-Miner [LZ05a] focused on programming rules, *e.g.*, function `b()` must be called after function `a()`. The rules are also extracted from the code history of real software systems, in which inconsistencies in these rules led to bugs. The tool also locates code that violates these rules and recommends transformations to fix them. Similar to refactoring approaches, both API usage and bug fix approaches search for very specific properties in code, *e.g.*, API calls and known patterns that would introduce bugs. Moreover, the recommended transformations are mostly extracted from the code history of the system under analysis. In our work, the transformations are considered as occasional but repetitive. Therefore, we require support for the application of these transformations *in situ*.

Finally, concerning a developer-defined sequence of transformations, some tools proposed to analyze the code under transformation to find other locations in which the sequence could be performed. LASE [MKM13] relies on code examples from the developer, *e.g.*, the source code before and after the developer fixed a bug in a method. The tool computes a collection of unmodified statements in the code examples. The tool then searches for methods containing similar statements by matching nodes in the AST. Critics [ZSPK15] relies on one code example. The tool also calculates unmodified statements from both old and new methods. However, Critics allows the developer to generalize program constructs in the modified code. For example, by manually generalizing a temporary variable, the tool will search for methods containing the same statements matching a temporary variable with any given name. Both tools relied on transformations related to bug patches, which generally comprise few and very localized transformations, *e.g.*, some additions and removals inside a method. In this thesis, we found examples of transformations that involve from adding statements in a method to modifying the hierarchy of classes, as shown in the example in Chapter 2.

Summary. Most of the approaches search for specific properties on source code, according to specific tasks, *e.g.*, correct a smell, update an API, fix a bug. The sequences of code transformations we work in Chapters 4 and 5 are system-specific, *i.e.*, these sequences may not be generalizable to other systems.

3.5 Design Smells and Negative Impact on Software Quality

Code transformations are often motivated by changes in requirements, and much less by code quality improvements, or system organization [Pér13, STV16]. Violations of design principles, known in the literature as design smells, might be introduced. These violations may not produce errors or affect behavior, however they might remain latent and negatively impact the quality of a system [GPEM09, SSS14]. In this section, we present examples of design smells we use in this thesis (Section 3.5.1), and we discuss the relevance of correcting design smells during software evolution (Section 3.5.2).

A *"smell"* is defined as a structure in the software that indicates a potential problem [PÍ11, SSS14]. The term was first introduced as *"code smell"*, limiting the range of analysis to source code [FBB+99]. Since then, other terms, such as *defect* and *flaw*, have been introduced in a similar manner.

Smells can occur at different levels of abstraction. Suryanarayana *et al.* [SSS14] classifies smells in two axis: (i) at the level of analysis: architecture, design (*i.e.*, micro-architectural), and implementation (*i.e.*, code); and (ii) at the level of impact: structural, and behavioral. In this thesis, we focus on structural *design smells*, defined as follows.

Definition 2 A structural *design smell* is a characteristic in the source code that indicate violations of fundamental *design principles* that negatively impact *design quality* [MGDM10, SSS14]. Design principles comprise the major elements in the object model, as proposed by Booch [Boo04]: Abstraction, Encapsulation, Modularization, and Hierarchy. Moreover, design quality properties that might be affected include understandability, maintainability, reusability, *etc*.

It is worth noting that, as an *indication* of a problem, it is expected that not all constructions detected as smells are in fact validated by developers. Fontana *et al.* [ADW$^+$15] classified false positive code smells and discovered that some smells might be imposed by the design, the framework, and optimizations, for example. There is also subjectivity on the automatic detection of smells [BBM10]. These results are supported by studies on developers' perceptions on smell detection results [SYA$^+$13, Yam14] Some particular structures may look like smell to an automated detection tool, but be considered valid by the developers under specific circumstances.

We now present examples of design smells we study in this thesis. We do not cover all the design smells proposed by Suryanarayana *et al.* [SSS14]. We recommend further reading on other design smells that are presented in their book. Both the definition of the smells and their aliases in the literature come straight from the book.

3.5.1 Examples of Design Smells

In this section, we present two examples of design smells we study in this thesis. We discuss how these smells manifest in source code. We detected the introduction of these smells during the evolution of Pharo ecosystem, and we proposed automatic correction for them in Chapter 7.

Scattered Functionality

The principle of modularization advocates the creation of cohesive and loosely coupled abstractions through techniques such as localization and decomposition [SSS14]. Scattered Functionality [GPEM09][1] arises when multiple modules are responsible for realizing the same high-level concern.

[1]also known as Broken Modularization [SSS14], or Misplaced Class [FBB$^+$99].

Scattered Functionality appears when a single concern is scattered across multiple components [GPEM09]. Both definition of component and high-level concern are general. We now discuss different interpretations of this smell.

Suryanarayana *et al.* [SSS14] consider modularization at the level of classes, *i.e.*, a class is a component; the smell manifests when methods in a class mostly access data from other classes. Similar scenario is defined in the Feature Envy code smell [FBB+99]. Methods that access this data ideally should have been localized in a single class. A high-level concern is then represented in a functional manner, *i.e.*, the access dependencies between classes express that these classes have the same concern.

At the package level, Fowler *et al.* [FBB+99] defines a misplaced class as the one contained in a package that contains other classes not related to it. Such *"relationship"* might be computed by cohesion metrics [AL11], by coupling metrics [Mar02], or by similarity of text extracted from identifiers and comments [Pal15].

Some quality attributes that can be negatively impacted by this smell include:

- **Understandability:** This smell negatively impacts the ability of developers to understand how a particular functionality is realized across multiple types or components.
- **Changeability and Extensibility:** Supporting modifications and extensions might become difficult since transformations might need to be performed across multiple components.
- **Reusability and Testability:** When one wants to reuse or test a particular functionality, one needs to use multiple types or components together instead of only one.

Unfactored Hierarchy

Unfactored Hierarchy[2] arises when there is unnecessary duplication among types in a hierarchy. Duplication manifests on methods with similar implementation, either the methods are exactly the same or they have enough similar fragments to be factored out.

Listing 3.1 presents a concrete example of Unfactored Hierarchy in GLAMOUR, one of the projects in the Pharo ecosystem. For comprehension purposes, we illustrate the code examples using a Java-inspired syntax. The entire code of the method under analysis is considered.

[2]also known as Orphan Sibling Method/Attribute [sis], Incomplete Inheritance [Bie06], Repeated Functionality [Are04], and Significant Sibling Duplication [inf12].

Listing 3.1: Example of a method under analysis in Glamour and an example of duplicated code in a subclass (highlighted in the second method).

```
class Model {
  public iconNamed(String iconName) {
    return Smalltalk.ui().getIcons().iconNamed(iconName);
  } //... }

class GLMUIThemeExtraIcons extends Model {
  // method with similar code
  public instVarRefactoringMenu(builder) {
    return (builder.newItem("Remove"))
        .setParent("Inst Var Refactoring")
        .setIcon(Smalltalk.ui().getIcons().iconNamed("removeIcon"))
        .setOrder(200);
  }  //... }
```

In this example, the class GLMUIThemeExtraIcons uses a fragment of code that is mostly implemented in its superclass, Model. A small variation consists in the method iconNamed(String), to which GLMUIThemeExtraIcons class has access, which requires an argument. One suggestion to remove this smell would be to replace the duplicated code in the subclass with an invocation to the method in the superclass, using "*removeIcon*" as an argument.

Some quality attributes that can be negatively impacted by this smell include:

- **Understandability:** Duplication within the hierarchy bloats the types with unnecessary code. It negatively impacts the ability of developers to understand the hierarchy and design.
- **Changeability and Testability:** Changing and/or testing code that is similar among types in a hierarchy requires replicating the same changes and tests across all those types.
- **Extensibility and Reusability:** Since common behavior is not well separated in the supertypes, developers might also duplicate code when introducing new subtypes in the hierarchy.

Several approaches and tools for clone detection have been proposed in the literature. These approaches are categorized by the representation of clones: as strings, tokens, abstract syntax subtrees, or semantic dependence graphs [JMSG07]. Zhang *et al.* [ZHB11] identified nine papers focusing on different methods to detect code duplication. In this thesis, we implement a tree-based heuristic approach that is specific to the case study we conducted with real developers.

3.5.2 Evolution of Design Smells

The negative impact of smells in software quality has been studied in literature. Most of related work focus on code smells as introduced by Fowler *et al.* [FBB+99]. The occurrence of code smells, such as God Class and Shotgun Surgery, are likely to increase the occurrence of bugs [LS07] and to decrease quality factors such as understandability and maintainability [DSA+04]. The occurrence of Duplicated Code, which is a more general case of Unfactored Hierarchy that we study in this thesis, although considered reliable by developers [MNK+02, KG08], indicated a maintainability decrease in a 20 years old industrial system [MNK+02].

Concerning the evolution on the occurrence of smells, we report, for each of the following studies: the dataset and the smells under analysis, and the main findings of each study.

Vaucher *et al.* [VKMG09] investigated trends of evolution on the occurrences of God Class smell. Two systems were analyzed, namely Xerces and Eclipse JDT. Most of the God Classes in these systems were introduced in the beginning of systems' lifecycle. 82% of these classes were playing roles in at least one of the following design patterns: Abstract Factory, Adapter, Observer, and Prototype. Additionally, developers mentioned that God Classes were as complex as the problems they addressed. Increasing the size of classes already detected as God Class seemed to be a common practice; developers constantly added new functionalities and data classes to God Classes. Moreover, refactoring suggestions as proposed by Fowler *et al.* [FBB+99] created new instances of God Class. The authors suggested transformations to correct instances of God Class that were specific in the systems under analysis.

Olbrich *et al.* [OCBZ09] analyzed a total of 76 selected revisions in two real software systems (Lucene and Xerces), and investigated phases of increase or decrease in the occurrence of code smells. The authors also showed that classes detected as God Class and Shotgun Surgery smells were changed more frequently in the code history of the systems under analysis. However, higher change frequency might be intrinsically related to the size of the classes, instead of a cause-effect relationship.

Peters and Zaidman [PZ12] investigated how often preventive maintenance is performed to remove the smells in seven industrial and open-source projects, small to large (119 to 821 classes). Five smells were investigated: God Class, Feature Envy, Data Class, Message Chain Class, and Long Parameter List Class. Feature Envy is more likely to be refactored in the systems under analysis, while God Class proved more difficult to be removed. The number of smells which persist during the system's lifecycle increase over time. Moreover, usually one or two developers remove more smells than their colleagues, and most instances were removed as a side effect of other maintenance activities, and not necessarily preventive maintenance.

Chatzigeorgiou and Manakos [CM14] analyzed two small systems (40 and 110 classes) and a total of 14 revisions. Four code smells were investigated: God Class, Feature Envy, Long Method, and State Checking. The study showed that: (i) there was an increase in the number of occurrences of all four code smells; (ii) the majority (89.8%) of these occurrences persisted until the last revision of the system, and more than a half of them (57.7%) existed since the first revision under analysis; and (iii) no preventive maintenance was realized in these systems; the eventual decrease in the occurrence of smells was result of adaptive maintenance, *i.e.*, by adding or removing functionalities.

Tufano *et al.* [TPB$^+$15] analyzed three software ecosystems, namely Android, Apache, and Eclipse, to understand when and why particular smells occurred in these ecosystems. Five smells were investigated: Blob Class, Class Data Should be Private, More than nine thousand commits were identified as responsible for introducing one of these smells. In at least half of the occurrences of smells were introduced when a code entity was added to the versioning system. Additionally, more than 80% of the smells were introduced when developers added features or improved existing features, *i.e.*, when they perform perfective maintenance. Smells are generally introduced in the last month before issuing a release, and developers were generally under heavy workload (in terms of number of commits) when they introduced the smells.

Summary. Smells tend to remain in the system since they are introduced. Developers are not usually concerned about the occurrence of smells. Although the quality of the system is compromised, the smelly code represent the best solution for the system under some circumstances. Preventive maintenance is not usually performed, and the correction of smelly code was caused by a side effect of adaptive maintenance, *i.e.*, by adding or removing functionalities.

3.6 Improving Code Transformations

In this section, we discuss approaches that suggest additional transformations when developers are transforming code.

Hayashi *et al.* [HSK06] proposed an approach to define rules for additional refactoring, *e.g.*, if code transformation is performed, under certain constraints, then recommend a sequence of transformations. The approach monitors three code transformations from the development tool, listed as follows.

- *Add Invocation* as argument of an existing invocation. The approach recommends to extract the added invocation to a temporary variable, *i.e.*, *Introduce Explain Variable* refactoring.

- *Copy-and-Paste Class C* to another class C_{copy}, which introduces Duplicate Code smell. The approach suggests *Extract Superclass* refactoring: (i) C and C_{copy} inherit a newly created superclass with the content of class C, then (ii) the implementations of C and C_{copy} are deleted.
- *Add Statement* in a duplicated class C_{copy}, *i.e.*, after performing *Copy-and-Paste Class*. The approach suggests *Extract Superclass*, followed by *Form Template Method* refactoring to maintain the modified behavior in C_{copy}.

These code transformations are stored in a database, in which the rules are evaluated. A database representation avoids performance issues caused by constantly checking design smells after the developer modifies code. Hayashi *et al.* [HSK06] evaluated the feasibility of the approach, however they did not evaluate whether the suggestions are accepted by real developers.

Liu *et al.* [LGS13] proposed an Eclipse plug-in which detects eight smells, including Duplicate Code, using existing smell detection tools. The smell detection is performed locally in the transformed code. However, Liu *et al.* [LGS13] do not propose a sequence of transformations to remove the smells. It is required for the developer to validate the smell and perform the transformations, either manually or using refactoring provided by the Eclipse IDE. The evaluation was conducted with 20 inexperienced developers and it showed that smells are solved more efficiently, and the introduction of new smells reduced by 50%.

3.7 Conclusion

In this chapter we showed that there are limitations in current approaches to support systematic code transformation, in particular:

1. the identification of system-specific sequences of code transformations, *i.e.*, they cannot be applied to other systems;
2. the definition of sometimes complex code transformations, *i.e.*, not only the addition or removal of lines of code;
3. the automation of code transformations that are not limited to a method's body, for example; and
4. the recommendation of additional transformations after, particularly, design smells might be introduced as a consequence of such systematic transformation.

In this thesis, the identification of sequences of transformations was performed semi-automatically, given its complexity, which we present in Chapter 4. Chapters 5 to 7 present our approaches to cover items 2 to 4, respectively.

4 Relevance of Systematic Code Transformation

Contents

4.1 Introduction

As presented in Chapter 3 in Section 3.2, previous researchers found evidences of systematic code transformation in real-world systems. Sequences of code transformations were discovered from the code history of the systems under analysis. These sequences were performed manually by developers, as some of the transformations are not fully supported by existing refactoring tools.

We reported approaches that identified patterns of bug fixes and API updates. Both activities can be (i) generic, *i.e.*, they can be applied to systems of different domains. Additionally, most of the sequences were (ii) simple, *i.e.*, they were composed of short sequences of additions and removals; and most of the sequences were (iii) performed locally, *i.e.*, they were generally applied to a single class or method at a time.

In this chapter, we report a study to investigate the existence of systematic code transformations on real cases of rearchitecting. The systems we analyzed underwent large evolution efforts, in which code transformations were applied to the systems at large. Consequently, we found instances of systematic code transformation that were:

- system-specific, *i.e.*, the sequences may not be generalized to other systems;
- sometimes complex, considering the transformations involved; and
- sometimes not-localized, *i.e.*, several methods or classes were transformed at each time.

The main contributions of this chapter are summarized as follows.

- We demonstrate the existence of systematic code transformations on real cases of large evolution effort.
- We manually validated the sequences of code transformations we found, to discuss the importance of their automated support.

Structure of the Chapter

Section 4.2 presents how we define sequences of code transformations. Section 4.3 presents our research questions and our experiment setting. Section 4.4 details the results of our investigative study. Section 4.5 discusses threats to validity of the study, and Section 4.6 concludes the study.

4.2 Defining Systematic Code Transformation

In this section, we present a real example of systematic code transformations. Similar to the one presented in Chapter 2, this example was applied on source code (Section 4.2.1). We use this example to discuss how we define sequences of code transformations in this thesis (Section 4.2.2), and to also discuss about the properties in these sequences that motivate their automation (Section 4.2.3).

4.2.1 Motivating Example

The transformations we present were extracted from PackageManager, a tool that manages package dependencies and versioning (similar to Maven). Listings 4.1 and 4.2 present code edition examples in two distinct classes, named GreasePharo30CoreSpec and SeasideCanvasPharo20Spec. We represent source code that was transformed in terms of added (+) and removed (–) lines.

Listing 4.1: Modified code in GreasePharo30CoreSpec

```
– public void platform() {
–     package.addPlatformRequirement("pharo");
–     package.addProvision("Grease–Core–Platform");
– }

+ public String[] platformRequirements() {
+     return { "pharo" };
+ }

+ public String[] provisions() {
+     return { "Grease–Core–Platform" };
+ }
```

Concerning the transformations involved, the developers removed a method named platform(). This method defines: (i) on which IDE configuration the current package depends, by invoking the method addPlatformRequirement(String); and (ii) the name of the package in its repository, by invoking the method addProvision(String).

Instead, the developers updated this definition so that each package "*only provides data and do not call methods*".[1] To accomplish this task, the developers added two methods, named platformRequirements() and provisions(). Both of them return an array of strings, containing the same arguments as in the platform() method. This new definition is more similar to a package manifest.

These transformations impact three methods of one class. Although these transformations seem simple, they were systematically performed in 19 distinct classes. Specifically, the transformations applied to all classes that extend the class PackageSpec and define a method named platform(). Other few classes, which are responsible for deserializing the package definitions, were transformed as well since the method platform() was removed. However, they were not transformed in a repetitive way and therefore they are not considered in this discussion.

Listing 4.2 shows the result of the same transformations, this time performed on the class SeasideCanvasPharo20Spec.

Listing 4.2: Modified code in `SeasideCanvasPharo20Spec`

```
- public void platform() {
-     package.addPlatformRequirement("pharo2.x");
-     package.addProvision("Seaside-Canvas-Platform");
- }

+ public String[] platformRequirements() {
+     return { "pharo2.x" };
+ }

+ public String[] provisions() {
+     return { "Seaside-Canvas-Platform" };
+ }
```

From both code examples, we observe that the code transformations themselves are the same, *e.g.*, a sequence of one *Remove Method* then two *Add Method* transformations. However, the sequence was applied to different code locations and, consequently, non-identical code was transformed in each class. Similar to an algorithm, a code transformation requires some input to be executed, *e.g.*, the signature of the method to be removed in our example. We call this information, the *parameters* of the transformation.

[1]We found this commit message in PackageManager's version control repository.

Definition 3 A *parameter* is an input that is necessary for a transformation to be performed. For example, to perform an *Add Method* transformation, one must provide the signature of the method, (optionally) the source code of the method, and the class to which this method will be added.

Table 4.1 summarizes the parameters in the examples as presented in listings 4.1 and 4.2.

Table 4.1: Parameters required to perform the transformations in the classes GreasePharo30CoreSpec and SeasideCanvasPharo20Spec. The parameter *"Modified Class"* is reused when removing and adding methods in this example.

Parameter	GreasePharo30CoreSpec (as seen in Listing 4.1)	SeasideCanvasPharo20Spec (as seen in Listing 4.2)
Modified Class	GreasePharo30CoreSpec	SeasideCanvasPharo20Spec
Removed Method	platform()	platform()
Added Method	platformRequirements()	platformRequirements()
Added Return Statement	{ "pharo" }	{ "pharo2.x" }
Added Method	provisions()	provisions()
Added Return Statement	{ "Grease-Core-Platform" }	{ "Seaside-Canvas-Platform" }

Some parameters are (i) similar in both transformations, *e.g.*, the signatures of the removed and added methods are the same in both examples of Table 4.1. Other parameters are (ii) non-identical, *e.g.*, the return statements in Table 4.1 vary from one class to the other one. Therefore, performing this sequence of transformations in a new location requires the developer to evaluate which parameters of the transformations should be changed in this location, and then to perform the transformations in the same order. These facts motivate the definition of system-specific sequences of code transformations, that can be configured by the developer, and then be applied automatically to other similar code locations.

4.2.2 Transformation Pattern Definition

The example in PackageManager displays a pattern, *i.e.*, a regular and repeated procedure, of code transformations that are performed systematically in this system. We call these discovered sequences of transformations, *transformation patterns*. The term is not related to design patterns because our sequences might be system-specific, *i.e.*, they might not be applicable to other systems. In this section, we define more precisely transformation patterns.

Program Representation

We represent programs in terms of code entities.

Definition 4 A *code entity* is a syntactically correct programming construct that is represented by: (i) a fragment of source code, and (ii) a corresponding subtree in the Abstract Syntax Tree (AST) that represents the software system [RL08,NNN$^+$13]. Alternatively, we also use the term *code location* to denote the location of a specific code entity in the source code of the system.

For example, a return statement is represented as an AST node (*e.g.*, ReturnNode) with one subtree specifying the value to be returned. This value might be an expression, a method call, a variable, or a literal value (*e.g.*, a string). Similarly, this returned value is represented as an AST subtree as well.

Transformation Representation

As discussed in Chapter 3 in Section 3.3, code transformations might be atomic and describe the addition and removal of code entities (level one), aggregated from level one transformations but still frequent (level two), or more complex and specific aggregations of levels one and two transformations (level three). In this thesis, we focus on the automation of level one and two transformations, because (i) one can compose them into more complex, level three transformations; and (ii) level one and two transformations are small and easy to implement and test.

Definition 5 A *transformation operator* is a code transformation that can be atomic or aggregated, *i.e.*, it considers transformations of levels *one* and *two*.

Definition 6 A *transformation pattern* is a sequence of transformation operators. Therefore, according to the categorization of Javed *et al.* [JAP12], the transformation pattern is a code transformation of level three.

To illustrate an example of the transformations we consider, we use again the PackageManager example. Listings 4.1 and 4.2 showed the result of the code transformations in a text based format. We represent the same example in terms of transformation operators in Pattern 4.2. These transformations were applied based on dependencies, *e.g.*, arguments and invocations, in a method named platform(), conveniently represented in Pattern 4.2 as M.

In this pattern, each step (lines 1 to 3) consists of a transformation operator. We consider that adding a method would optionally also include adding its content, *e.g.*, a return statement. Therefore, in this representation, the *Add Method* is a level two transformation. Moreover, each transformation operator requires some parameters to be specified, *e.g.*, class C, signatures of methods M, M', and M''. We provided two examples of the parameters for this pattern in Table 4.1.

PATTERN 4.2: PackageManager's transformation pattern.

Applied to: 19 classes.

1. *Add Method* M' named *"platformRequirements"* in C, containing
   a return statement with the argument of *"addPlatformRequirement(String)"* in M
2. *Add Method* M" named *"provisions"* in C, containing
   a return statement with the argument of *"addProvision(String)"* in M
3. *Remove Method* M

4.2.3 Properties of Transformation Patterns

We define three important properties of a transformation pattern. These properties highlight the need to study, document and automate the patterns. The properties are described as follows and they are evaluated on real systems in Section 4.3.

Frequency denotes the number of occurrences of the pattern in a single revision of the system. For example, the pattern in PackageManager was applied in 19 distinct classes, therefore its frequency is 19. Frequency is the most important property to propose some automation of transformation patterns, *i.e.*, the more frequent a pattern is, the more tedious its manual application will be.

Complexity relies on two characteristics. The *number of transformation operators* concerns how many transformation operators have to be repeatedly applied. Our working example presents three transformation operators, as shown in Pattern 4.2. Moreover, the *number of parameters* concerns how many parameters have to be considered in each repetition of the pattern. Table 4.1 presents six parameters for our working example. In our study, we found patterns that are not very frequent, however they are rather complex.

Recurrence relates to the occurrence of a pattern on several revisions of the system. This indicates that the pattern is rather complex to be applied (relating to the previous property), or it is difficult for developers to find all the code entities in the system that should be transformed.

Finally, it must be reinforced that, contrary to refactoring transformations, we do not impose the behavior preservation of the source code. We observed that such repetitive transformation usually makes sense in the context of punctual effort to improve the organization of a system, *e.g.*, a rearchitecting. It seems less likely that these patterns can be found in normal, day-to-day, maintenance activity. Therefore, one must expect and accept that the code will pass through an unstable state during a systematic code transformation.

4.3 Investigative Study Setting

In this section, we study systems that underwent a rearchitecting effort in the past, in order to find transformation patterns that are specific to these systems. Our goal is to understand how these patterns were applied in real conditions. We first present the proposed research questions in Section 4.3.1. We present the systems under study in Section 4.3.2. We discuss our methodology for finding transformation patterns in Section 4.3.3. We present the transformation patterns we found, answering RQ4.1 in Section 4.4.1. Then, sections 4.4.2 to 4.4.4 present the results we use to answer our remaining research questions.

4.3.1 Research Questions

We propose research questions to understand the importance of automated support in the application of transformation patterns. We restrict our study to system specific code transformations. We presented one example of transformation pattern in Section 4.2.1. As discussed in Chapter 3 in Section 3.2, there are evidences in the literature on the existence of systematic code transformation. However, there is a lack of approaches that identify system-specific and sometimes complex transformation patterns. Considering this specific context, we propose a main research question:

RQ4.1 *Can we identify system-specific transformation patterns in other systems?*

Assessing Transformation Patterns. We propose RQ4.1 to demonstrate the generality of the problem. To complement this research question, we also evaluate potential properties of the transformation patterns that motivate some automated support in their application. Note that we will not further formalize our research questions (formal hypothesis) or formally test them. All that is required in this study is proof of existence in various systems. We describe the complementary research questions as follows.

RQ4.2 *Are transformation patterns applied to all of the transformation opportunities?* We investigate whether the transformation patterns were applied to all of the code entities they were supposed to.

RQ4.3 *Are transformation patterns applied accurately in all code locations?* Given that a transformation pattern is a sequence of transformation operators, we investigate whether all of the operators were applied in each occurrence of the pattern.

RQ4.4 *Are transformation patterns applied over several revisions of the system?* We investigate whether the patterns were applied at once or over several revisions.

4.3.2 Target Systems

The dataset is based on previous work on real software systems that underwent rearchitecting [SVA14]. We added to this list systems that underwent a rearchitecting in our research group. In total, we have two Java systems and five Pharo[2] systems. Table 4.2 summarizes descriptive data about our dataset.

Eclipse went through a considerable rearchitecting to integrate the OSGi technology. We focused in the user interface plugin, which was separated into five new plugins in the version 3.0.

JHotDraw is a framework for technical graphics. Its rearchitecting aimed at specializing the interface of color spaces.

PetitDelphi is a parser for Delphi that has been enhanced to generate an AST from a tokenized tree. The restructuring aimed at pruning the generated AST nodes.

PetitSQL is a parser for SQL. It is built using a parser library called PetitParser [RDGN10]. Its rearchitecting focused on correcting API usage of the grammar.

PackageManager is a package management system for Pharo. Its rearchitecting focused on changing the interface to access package metadata.

GeneticAlgorithms is a small project that applied a specific type of genetic algorithm (*e.g.*, NSGA-II). It was rebuilt to allow different implementations of selection, crossover, and mutation algorithms.

Telescope is a visualization framework for Pharo. It went through series of transformations to specialize visualization builders.

4.3.3 Discovering Transformation Patterns

In this study, the identification process was mostly manual and *ad-hoc*. We benefited from some automated support in the beginning, *i.e.*, when computing code transformations from two versions of a rearchitecting. However, we inferred the sequence of code transformations manually by analyzing the transformed code.

It is worth noting that we could have used identification approaches in literature that we presented in Chapter 3 in Section 3.2. As discussed in that chapter, identification approaches retrieve simple sequences of transformations, *e.g.*, a short sequence of additions and removals inside a method. From our first example of systematic code transformation in Chapter 2, which repeatedly adds new

[2]http://pharo.org/

Table 4.2: Size metrics of our dataset. The systems are divided in two groups: Java and Pharo systems. Each line describes a rearchitecting between two versions. Metrics are shown in pairs (before and after the rearchitecting).

	Packages	Classes	KLOC
Eclipse-UI 2.1 / 3.0	68 / 118	2253 / 3329	185 / 277
JHotDraw 7.4.1 / 7.5.1	39 / 41	614 / 665	59 / 66
PetitDelphi 0.210 / 0.214	7 / 7	313 / 296	8 / 9
PetitSQL 0.34 / 0.35	1 / 1	2 / 2	0.3 / 0.4
PackageManager 0.58 / 0.59	2 / 2	117 / 120	2.5 / 2.3
GeneticAlgorithms 0.1 / 0.6	*1 / 3*	*15 / 20*	*0.5 / 0.6*
Telescope 0.219 / 0.272	*7 / 10*	*43 / 49*	*1.5 / 1.4*

classes and methods in the system, our goal was to find more complex transformation patterns. In this study, the proof of existence of transformation patterns is considered more important than the identification process.

Our identification process was done in four steps, described as follows.

First, we automatically compute code transformations between two versions of a rearchitecting, in terms of level one transformations (*i.e.*, added and deleted code entities). For this step, we used the diff calculator provided by Eclipse for Java systems, and also provided by the Torch tool [UGDD10] for Pharo systems.

Second, we automatically group sets of transformations that have the same type. In our example in Section 4.2.1, we found a set of *Remove Method* transformations, applied to methods with the same name, *e.g.*, platform(). These group sets could be the seed for candidate transformation patterns. Separately, we also found sets of *Add Method* transformations, also applied to methods with the same name, *e.g.*, platformRequirements().

Third, we manually identify the set of transformation operators. This step is manual because it requires identifying similar characteristics between the modified code entities. These characteristics might be, for example:

- an identical parent AST node, *e.g.*, both *Remove Method* and *Add Method* groups were applied to the same classes. We presented two of these classes in Section 4.2.1.
- an identical child AST node, *e.g.*, the argument "*pharo2.x*" in the removed method platform() is the same in the added method platformRequirements().
- a similar child AST node, *e.g.*, the signature of the added method platformRequirements() is derived from the invoked method addPlatformRequirement(String) in the removed method platform().

These are evidences that *Remove Method* platform() and *Add Method* platformRequirements() might merge into a new group set. Other characteristics proved to be more complex to identify in other systems. Additionally, these characteristics must at least repeat themselves in other occurrences, to justify the discovery and/or automation of a transformation pattern. The result of this step is a sequence of groups of transformations that were applied to non-identical but related code entities.

Finally, the fourth step consists in identifying an application condition that expresses why these code entities were systematically transformed together.

Definition 7 An *application condition* is an expression that selects, from all the entities in a system (*e.g.*, classes, methods, *etc.*), which ones must be transformed by a transformation pattern.

This step was also manual, *i.e.*, we also tried to infer which common characteristics in the code entities made them be systematically transformed. In our working example, the condition selected classes in a specific hierarchy (*e.g.*, the superclass PackageSpec) and it relied on the existence of a method named platform() which invokes both methods addPlatformRequirements(String) and addProvisions(String).

This identification process is tedious due to the huge amount of transformations under analysis, and it is error prone because the condition is sometimes hard to identify. In this study, there are cases in which the developers of the systems themselves helped to define the application condition properly. We come back to this point as a threat to validity.

It is worth mentioning that we manually extracted an application condition to better understand the transformation patterns. More specifically, we investigate whether this condition includes all the code entities that were actually transformed in Section 4.4.2, to answer RQ4.2. Later in this thesis (Chapter 6) we propose an approach to find locations in code where to apply a transformation pattern. However, we noticed that a general automated support to discover a definitive application condition, *e.g.*, the pattern was applied to all classes extending PackageSpec and having a method named platform(), was too complex for the systems we studied and therefore is out of scope of this thesis.

4.4 Investigative Study Results

4.4.1 The Patterns (RQ4.1)

In this section, we answer RQ4.1. We identified a total of nine transformation patterns in five out of seven systems under analysis. We identified more than one pattern in two systems, namely Eclipse and PackageManager.

We describe each transformation pattern as follows. We describe them using the format we proposed in Section 4.2.2. Additionally, we also describe the application condition we inferred for these patterns.

Eclipse's (first) transformation pattern

In Eclipse, we identified a pattern related to modularizing the Action hierarchy. Most subclasses of Action were moved from workbench plugin to ide plugin. Because of that, all of the invocations to methods of WorkbenchMessages had to be replaced by invocations to methods of a new class, called IDEWorkbenchMessages.

PATTERN 4.3: Eclipse's (first) transformation pattern

```
Condition: ∃ class C ∈ ui.workbench that extends jface.Action
1. Move Class C to plugin ui.ide
   ∃ methods $M \in C$, $M_W \in$ WorkbenchMessages and $M$ invokes $M_W$
2.    Add Static Method $M'_W$ to IDEWorkbenchMessages
3.    Move Statements of $M_W()$ to $M'_W()$
4.    Replace Invocation to $M'_W$ in method $M$
      by an invocation of IDEWorkbenchMessages.$M'_W()$
```

We see in this example a complex transformation pattern. It consists of four operators that impact two classes in each sequence of transformation operators (*e.g.*, C in line 1, and IDEWorkbenchMessages in line 2 in Pattern 4.3). This pattern illustrates a situation that we found often: some transformation patterns have internal conditions.

Eclipse's (second) transformation pattern

In Eclipse, we found another transformation pattern related to the use of the SafeRunnable abstract class. The method handleException(Throwable) is originally responsible to open a message dialog with an exception message. In the class documentation, developers advised: "*This class can open an error dialog and should not be used outside of the UI Thread*".

The pattern consisted in discovering all classes that extend SafeRunnable and override the method handleException(Throwable), and further remove these overriding methods.

PATTERN 4.4: Eclipse's (second) transformation pattern

```
Condition: ∃ class C extending SafeRunnable
    and overriding method handleException(Throwable)
1. Remove Method handleException(Throwable) in C
```

This pattern is very short, *i.e.*, it has only one transformation operator in Pattern 4.4. However, it was mostly applied to anonymous classes which are hard to manually inspect. We come back to this discussion in Section 4.4.4.

JHotDraw's transformation pattern

The rearchitecting in JHotDraw was applied to color spaces hierarchy, which extends AWT. All ColorSpace classes must implement a new interface called NamedColorSpace (line 1 in Pattern 4.5), which has only one method, named getName() (line 2). The transformation also includes the use of a Singleton design pattern (lines 3–4). Because of the design pattern, all direct instantiations must be replaced by an invocation to the method getInstance() (line 5).

PATTERN 4.5: JHotDraw's transformation pattern

```
Condition: ∃ class C that extends ColorSpace
1. Add Interface NamedColorSpace to C
2. Add Method getName() in C
3. Add Private Attribute instance in C
4. Add Static Method getInstance() in C
   ∃ method M that invokes new C()
5.    Replace Statement new C() by invocation to C.getInstance()
```

It is worth noting that, similarly for the two previous patterns, this one impacts not only the class extending ColorSpace but also all the classes that instantiate this class. We come back to this pattern in Section 4.4.3.

PetitDelphi's transformation pattern

For each grammar rule defined in class PDDelphiSyntax, PetitDelphi systematically creates a node in the resulting AST in subclass PDDelphiParser. When a rule is a disjunction of other rules (*e.g.*, a type is either a class or an interface), the rule causes the creation of two nodes in the AST: (i) one for the choice (TypeDeclaration) and (ii) a unique child for the actual node (which is either a Class- or an InterfaceDeclaration).

This AST generation was considered undesirable and the entire infrastructure was modified to suppress the creation of the intermediary node (TypeDeclaration). The pattern then removes the method which creates this node (line 1 in Pattern 4.6) in the subclass and the class representing the node (line 2).

PATTERN 4.6: PetitDelphi's transformation pattern

```
Condition: $\exists$ method $M \in$ PDDelphiSyntax and $M$ is a disjunction of other rules
1. Remove Method $M$ in PDDelphiParser
   $\exists$ class $C$ that $M$ instantiates (new C())
2.    Remove Class $C$
```

This pattern is not complex in itself. It removes the method and the class that represents the intermediary node. In spite of that, the pattern is difficult to apply entirely due to the difficulty of finding all the instances of disjunction rules.

PetitSQL's transformation pattern

PetitSQL features two classes (ASTGrammar and ASTNodesParser), the second inheriting from the first. A treatment is done by calling methods of the subclass which could return collections of elements. Some of these elements have to be filtered out in the treatment. The transformation consisted in overriding the existing withoutSeparators() method (lines 1–3 in Pattern 4.7), then removing the filtering based on the Collections' API (line 4).

PATTERN 4.7: PetitSQL's transformation pattern

```
Condition: $\exists$ method $M_P \in$ ASTNodesParser
      and $\exists$ method $M_G \in$ ASTGrammar
      and $M_P$ invokes $M_G$
      and $M_P$ then invokes Collection.filter()
1. Add Method $M'_G$ in ASTNodesParser (overriding $M_G$)
2. Add Reference to super in $M'_G$
3. Add Invocation to withoutSeparators() in $M'_G$
4. Remove Invocation to Collection.filter() in method $M_P$
```

This pattern has a complex application condition, with two parameters that depend on each other, *i.e.*, the method in which the filtering will be removed, and the recently added method that will invoke the new filtering API. This pattern was the one for which developers helped us define an application condition.

PackageManager's (first) transformation pattern

In PACKAGEMANAGER, packages are represented as data objects extending PackageSpec. The developers decided that packages should not be modified with setter methods. Other classes are also affected, such as PackageVersion and Dependency. In this system, we found four transformation patterns, all of them are related to the same modification.

In this first pattern, all subclasses of PackageSpec have a method (*e.g.*, dependencies()) which calls setter methods that were removed. This method should now create an array and represent the dependencies as associations between the name of the package and its corresponding version.

PATTERN 4.8: PackageManager's (first) transformation pattern

```
Condition: ∃ class C that extends PackageSpec
    and ∃ method M in C named 'dependencies'
    and ∃ statement S invoking addDependency(String) with argument A
1. Remove Statement S
2. Add Return Statement containing:
    an association with A and the result of self.getVersion()
```

PackageManager's (second) transformation pattern

The second pattern in PACKAGEMANAGER is our working example in this thesis. We first described the transformation operators involved in Section 4.2.1, and we discussed about the classes to which this pattern was applied in Section 4.3.3. We present the entire description of this pattern, including an application condition.

PackageManager's (third) transformation pattern

The third pattern in PACKAGEMANAGER affects the representation of packages as well. This time, packages specified the repository in which they were stored by invoking the method addRepository(String). These packages should now just return the repository path as a string.

PackageManager's (fourth) transformation pattern

As opposed to the previous PACKAGEMANAGER patterns, the fourth one does not affect the hierarchy of package representation. In this pattern, developers updated

PATTERN 4.9: PackageManager's (second) transformation pattern

```
Condition: ∃ class $C$ that extends PackageSpec
    and ∃ method $M$ in $C$ named 'platform'
    and ∃ statement $S$ invoking addPlatformRequirement(String) with argument $A'$
    and ∃ statement $S$ invoking addProvision(String) with argument $A''$
1. Add Method $M'$ named "platformRequirements" in $C$
    containing a return statement with $A'$
2. Add Method $M''$ named "provisions" in $C$
    containing a return statement with $A''$
3. Remove Method $M$
```

PATTERN 4.10: PackageManager's (third) transformation pattern

```
Condition: ∃ class $C$ that extends PackageSpec
    and ∃ method $M$ in $C$ named 'repositories'
    and ∃ statement $S$ invoking addRepository(String) with argument $A$
1. Remove Statement $S$
2. Add Return Statement containing $A$
```

invocations to methods of class Dependency when a constraint to a specific version of a package is needed. Invocations to addFixedVersionConstraint(String) were removed (line 1 in Pattern 4.11). Performing this invocation now requires an instantiation of FixedVersionConstraint (line 2).

PATTERN 4.11: PackageManager's (fourth) transformation pattern

```
Condition: ∃ method $M$ instantiating class Dependency in a temporary variable var
    and ∃ statement $S$ invoking addFixedVersionConstraint(String) with argument $A$
1. Remove Statement $S$
2. Add Invocation to var.withConstraint(Constraint), containing
    new instance of FixedVersionConstraint with $A$ as argument
```

GeneticAlgorithms and Telescope

We must also report that we studied two other systems for which we could not identify any patterns matching our definition: GENETICALGORITHMS and TELESCOPE. Because our method for identifying transformation patterns is mostly manual, we do not claim that there are no patterns in these systems. Equivalently, we do not claim that the patterns we found are the only ones in the other systems. Extracting transformation patterns from code history is not an easy task. We do not see this fact as a serious threat. We did not claim that the use of transformation patterns is inherent to the rearchitecting process, but only that it is likely to happen.

Summary. We identified transformation patterns in five out of seven systems. These systems use two different programming languages, and our study analyzed only one specific version of each system. We identified more than one pattern in two systems.

4.4.2 Are transformation patterns applied to all of the transformation opportunities? (RQ4.2)

With the application condition in a transformation pattern, we basically count, in the entire system, how many code entities match the condition. Then, we count the number of code entities that were actually transformed by the pattern. We count an occurrence even when not all of the transformation operators were performed (see RQ4.3), and even if the pattern was applied in more than one version of the system (see RQ4.4). We expect that developers, in lack of automated support, might have forgotten to apply the transformation patterns to all of the candidate code entities. Table 4.3 summarizes the results.

Table 4.3: Number of potential occurrences of the transformation patterns and number of actual occurrences. The number of occurrences in parenthesis is the frequency observed in the revisions under analysis (see Section 4.4.4 for recurrence results).

Transformation patterns	Entities matching condition	Pattern occurrences
Eclipse I	34	26
Eclipse II	86	(70)72
JHotDraw	9	9
PetitDelphi	21	(15)19
PetitSQL	6	6
PackageManager I	50	50
PackageManager II	19	19
PackageManager III	64	64
PackageManager IV	7	7

Most of the transformation patterns were applied in all of the transformation opportunities. This result seems natural for patterns with low frequency, such as JHotDraw and PetitSQL, for example. However, the same happens with PackageManager, which has some of the most frequent patterns. This behavior was motivated by the *Remove Method* transformations that broke the code, therefore the developers had to systematically correct the system to make it run again.

This obligatory characteristic relates to the PetitDelphi case. In this system, some potential code entities were not transformed initially in the first revision. According to the developers, transforming all of the code entities at once was not part of the rearchitecting effort they were conducting, and therefore they chose to leave it for later work. They actually ended up applying all the possible occurrences in later revisions (see RQ4.4). Along with PetitDelphi, Eclipse's transformation patterns were not applied to all possible occurrences as well.

Summary. Two out of nine transformation patterns were not applied to all the opportunities matching the application condition. When the patterns covered all the opportunities, this fact was due to their low frequency, or because the pattern consisted of a systematic and corrective task.

4.4.3 Are transformation patterns applied accurately in all code locations? (RQ4.3)

We want to identify transformation pattern occurrences in which not all of the constituting transformation operators were applied. We expect that developers might forget to apply some transformation operators because (i) the pattern is complex in terms of the number of operators involved; and/or (ii) the pattern affects more than one code entity in each occurrence, which requires the developer to constantly change the parameters of the transformations.

From all the patterns in our study, only the one in JHotDraw was not accurately applied. In this pattern, all subclasses (a total of nine) of ColorSpace must (i) implement a new interface named NamedColorSpace, (ii) consequently implement a method named getName, which is declared in this interface. Additionally, the transformation includes (iii) the implementation of the Singleton design pattern. In terms of number of steps, the pattern in JHotDraw is not the most complex pattern in our dataset. However, this transformation pattern impacts several classes. Specially when applying Singleton, the transformations do not only impact the subclass, but also all of the classes in the system that directly instantiate this subclass.

In JHotDraw, none of the classes implement the Singleton design pattern accurately. All of them missed changing the constructor accessibility to private. In fact, there are direct instantiations (via new) of three classes (out of nine concerned by this transformation). Particularly two classes do not implement Singleton at all, and one class does not extend NamedColorSpace. For this particular case, there are no instantiation to these three classes in the system. It is possible that these classes are not active in the system, and therefore should be removed in the future.

Summary. In one out of nine transformation patterns, not all of their transformation operators were applied. There does not seem to be a correlation between this fact and the number of transformation operators, nor with the number of occurrences of the pattern.

4.4.4 Are transformation patterns applied over several revisions of the system? (RQ4.4)

We select the transformation patterns that were not applied in all of the transformation opportunities in the first revision. Except for Eclipse and PetitDelphi, all of the patterns were applied in one revision. As discussed in RQ4.2, these patterns were the ones (i) with least frequency, which is expected; or (ii) they were the ones that initially introduce error in the code.

For this analysis, we select five revisions from Eclipse and PetitDelphi, including the rearchitecting one. Then we repeat the same analysis from RQ4.2, *i.e.*, we counted the number of code entities that are transformed by the pattern in each revision. Table 4.4 describes the revisions under analysis in Eclipse and PetitDelphi. For each revision, we accumulatively count the number of entities that applied the pattern, whether they applied it accurately (see RQ4.3) or not.

Table 4.4: Selected revisions with the number of occurrences of a transformation pattern for each pattern.

System	#Revision	Date	Occurrences
	3.0	25/06/04	70
	3.1	27/06/05	71
Eclipse II	3.2	29/06/06	72
	3.3	25/06/07	72
	3.7	13/06/11	72
	210	19/11/14, 14:52	15
	211	19/11/14, 18:56	17
PetitDelphi	212	26/11/14, 18:17	18
	213	03/12/14, 18:23	18
	214	22/12/14, 15:55	19

The second pattern in Eclipse consists in a single operator (remove an overriding method). Although modern IDEs such as Eclipse have facilities that would allow one to discover all the possible candidates for applying this transformation pattern (*e.g.*, search all the classes that override a method with a given name), it took two years and three revisions to go from 70 to 72 occurrences and, after seven years, there were still 14 ($86 - 72$, see Table 4.3) entities left.

The pattern in PetitDelphi took five revisions to be applied to four more candidate entities. These revisions extended over one month, however its rearchitecting demanded four hours per week.

Additionally, the first pattern in Eclipse was initially applied in 26 classes at the first revision (*e.g.*, from version 2.1 to 3.0). This pattern consisted in moving Action classes to another component and replacing invocations to a new class of this component. Between versions 3.2 and 3.3, a total of 28 Action classes were added in the ide component. These classes did not have to replace invocations because they invoke this class directly. However, we did not succeeded in obtaining a condition for this continuous addition. Therefore, we do not show results for this pattern in Table 4.4. This Eclipse case shows that *the result of this pattern* continued to be observed even when the pattern itself was not applied anymore.

Summary. Two out of eleven transformation patterns were applied in not one but several revisions. This fact might be related to the perfective maintenance nature of their transformations, *i.e.*, not applying the transformation pattern in all the occurrences did not have impact on the execution of the systems.

4.5 Threats to Validity

4.5.1 Construct Validity

The construct validity is related to whether the evaluation measures up to its claims. In our study, we evaluate whether the transformation patters were applied to all of the transformation opportunities (RQ4.2). This evaluation depends on the condition attached to the patterns. In most of the cases, we defined the application condition. There is a risk that the condition we defined might be too extensive, *i.e.*, the condition will select more transformation opportunities than desired.

For Pharo systems, we relied on the developers of the target systems. In PetitDelphi and MooseQuery for example, the condition was not clear at first. The developers of these systems helped us define the application condition correctly. Even so, in PetitDelphi we identified missing transformation opportunities, because the developer wanted to focus on a specific part of the system at a given moment. In Eclipse (second pattern), there are also missing opportunities on which the developers came back later. We consider them evidences that we described the condition correctly.

4.5.2 Internal Validity

The internal validity is related to uncontrolled factors that might impact the experiment results. In our study, the developers of four of the systems under analysis are members of our research group. Indeed, the fact that some of us knew systematic code transformation cases was a big help. However, it must be noted that the patterns we found occurred before our study, *i.e.*, their identification was *post-mortem* and therefore was not influenced by our analysis. Our participation in the development only helped us to re-discover them.

4.5.3 External Validity

The external validity is related to the possibility to generalize our results. First, most of the systems in our dataset are small (RQ4.1). One may argue that the less entities a system has, the less occurrences a pattern will present. Moreover, it would be easier to understand the small system and manual systematic transformation would not have much effort. However, the patterns in PackageManager, considered as small (see Table 4.2), indicate that the size is not an issue. The transformation patterns we found in them were one of the most repetitive ones, with 64 and 50 occurrences (see Table 4.3).

4.6 Summary

Evolution is an important activity in the software system lifecycle. It can take several forms such as bug correction, addition of new functionalities or, more occasionally, substantial modifications of the system as a whole. During a rearchitecting, developers sometimes perform sequences of code transformations to distinct but similar code locations. We called these sequences, transformation patterns.

Due to the repetitive nature of these transformations, applying them manually is a tedious process. The manual application might be a complex task, either because of the number of transformations involved or the selection of code entities that must be transformed. And finally, this task might also be error-prone. Developers might miss code entities that must be transformed, or they might not perform all the transformations defined in the sequence.

In this chapter, we investigated rearchitecting cases from real-world systems, small to large, and we found instances of transformation patterns. The evaluation leads to the conclusion that systematic code transformation is a phenomenon that occurs during a rearchitecting process. The transformation patterns we found are language independent, *e.g.*, we studied Java and Pharo systems, but they are specific to the systems in which we found them.

The results presented in this chapter show that transformation patterns were not always applied to all the entities that should be transformed. In some cases, developers did not perform all the transformations in the sequence, or all the transformations were not performed in one shot but over several revisions.

In that respect, one would benefit from automated support to avoid omission due to the manual and repetitive application of these patterns. In this thesis, we propose automation to apply transformation patterns in two ways. First, to automatically apply sequences of code transformations composed by the developer. And second, to recommend code entities that should be transformed after a sequence of transformations is performed. The next two chapters present solutions to perform such activities to reduce the manual work from developers.

5 Supporting Systematic Code Transformation

Contents

5.1 Introduction

In the previous chapter, we found instances of systematic code transformation that were manually performed by developers. We presented evidence that sequences of code transformations were sometimes complex and error-prone to be applied manually. As presented in Chapter 3 in Section 3.4, most approaches that provide automation for code transformations were applied to perform small sequences that are most likely to modify one method at each time. Large evolution efforts such as the one we presented in Chapter 2 are not supported. There is a lack of approaches that can handle system-specific, sometimes complex, and not localized sequences of transformations.

In the previous chapter, we found sequences of code transformations that were performed manually and systematically, *e.g.*, to up to 72 occurrences. We propose a solution to automate their application that consists in (i) allowing the developer to compose a sequence of small code transformations; then (ii) providing support to apply this sequence in similar but non-identical code locations. This automation would reduce errors of omission caused by repetitive and manual transformation, *e.g.*, missing transformation operators as discussed in Chapter 4 in Section 4.4.3. We present an example of real transformations in Section 4.2.

In this chapter, we report the proof-of-concept implementation of MacroRecorder. This prototype tool allows the developer to automatically perform transformation patterns. Figure 5.1 describes our approach.

Using our tool in practice, the developer performs the code transformations once (Figure 5.1.1). The tool *records* (Figure 5.1.2) these transformations as a new *macro*. The developer further selects, in the code browser, a different code location (Figure 5.1.3) in which the tool must replay the recorded transformations. The developer optionally *configures* the transformations to apply them in this new location. The tool attempts to apply the transformations (Figure 5.1.4), by adapting the parameters of the pattern as they were calculated in the recording step. If successful, MacroRecorder *replays* the transformations automatically.

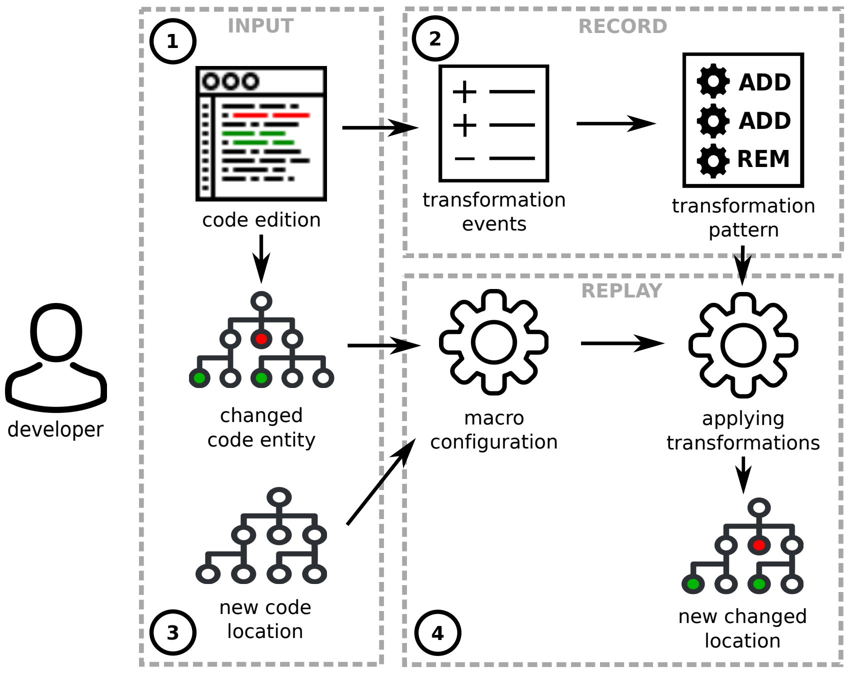


Figure 5.1: Overview of our approach. The developer provides a sequence of transformations (1) that will be recorded (2) from the development tool, and a new code location (3) in which this sequence must be replayed (4).

The main contributions of this chapter are summarized as follows.

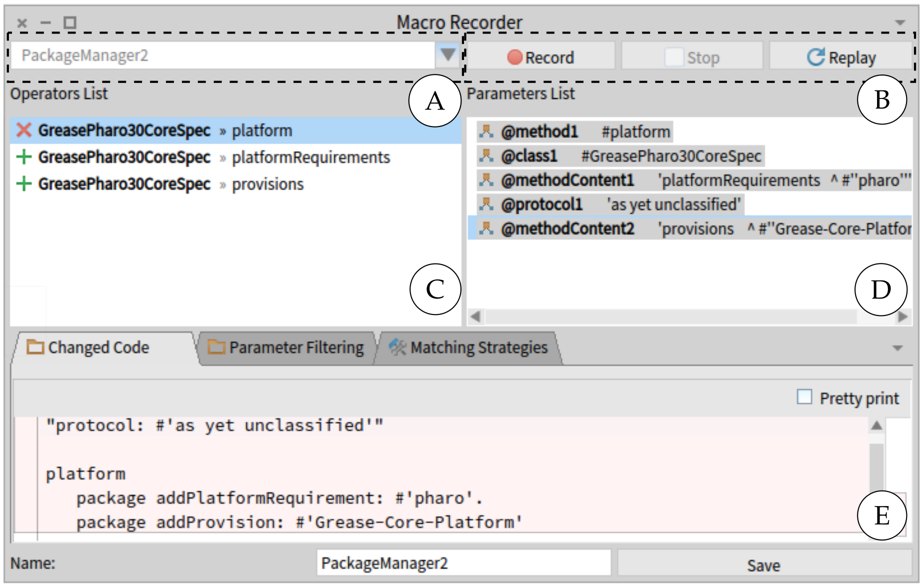
- We propose an approach to record and replay sequences of code transformations.
- We validate this approach on the system-specific sequences we discovered in Chapter 4.

Structure of the Chapter

Section 5.2 presents how MacroRecorder works from the point of view of a potential user. Section 5.3 presents implementation details. Section 5.4 presents our evaluation to validate the tool. Section 5.5 details the results of the validation study. Section 5.6 discussed threats to validity of the study, and Section 5.7 concludes the study.

5.2 MacroRecorder in Action

Figure 5.2 depicts the main user interface of MacroRecorder. The panel (A) shows a list of transformation sequences that were already recorded by the user. The toolbar in panel (B) shows buttons to start and stop recording code transformations, and to replay a transformation pattern. Panel (C) presents the current sequence of transformations as recorded by the tool. Panel (D) presents the parameters of the transformation sequence, *i.e.*, an aggregated list of the parameters from all the transformation operators. Finally, the tab panel (E) shows (i) the result of the selected transformation operator, (ii) a list of filters options for displaying parameters in panel (D), and (iii) a list of matching strategies that will be discussed in Chapter 6. We use the PackageManager example in Section 4.2.1 to show how MacroRecorder works.



GreasePharo30CoreSpec » platform
GreasePharo30CoreSpec » platformRequirements
GreasePharo30CoreSpec » provisions

@method1	#platform
@class1	#GreasePharo30CoreSpec
@methodContent1	'platformRequirements ^ #''pharo'''
@protocol1	'as yet unclassified'
@methodContent2	'provisions ^ #''Grease-Core-Platfor

```
"protocol: #'as yet unclassified'"

platform
   package addPlatformRequirement: #'pharo'.
   package addProvision: #'Grease-Core-Platform'
```

Figure 5.2: MacroRecorder overview. The panel (C) shows three transformation operators (one removal and two additions), and panel (D) shows five parameters (see text for details on other panels).

5.2.1 Recording Code Transformations

A session with our tool starts with the user pressing the "*Record*" button (Figure 5.2, panel B). Apart from starting and stopping the recording, this process is transparent to the developer. The developer performs the code transformations normally by manually editing the source code, or with the automated support (*e.g.*, refactoring) from the IDE, while the tool records the transformations in the background. After performing the transformations, the developer stops recording ("*Stop*" button in Figure 5.2, panel B).

As a result of this process, Figure 5.2.C depicts the sequence of transformation operators that were recorded in our example. MacroRecorder recorded three transformation operators in PackageManager: *Remove Method* platform() from the class GreasePharo30CoreSpec, *Add Method* platformRequirements() and *Add Method* provisions(), all of them performed in the same class.

After saving this sequence of transformations (Figure 5.2.E, "*Save*" button), MacroRecorder (i) creates a new macro from these transformation operators, and (ii) stores this macro in the development IDE for later use.

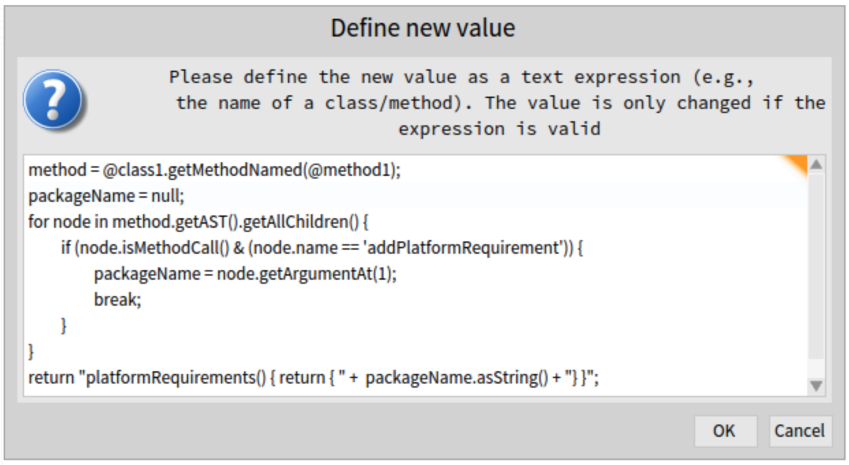
Definition 8 A *macro* is an occurrence of a transformation pattern that was recorded by the developer for later application. At this point, we make the distinction between transformation pattern, as a phenomenon of systematic code transformation that was observed, and a macro which can effectively transform the code.

5.2.2 Configuring Code Transformations

As shown in Figure 5.2.D, MacroRecorder stores the parameters of the macro. To replay the macro in a new code location, one must modify the values of each parameter according to this new location. Currently, MacroRecorder proposes automatic and manual configurations of transformation patterns. The automatic configuration will be explained in details in Chapter 6.

In the manual configuration, the tool allows the developer to specify an expression that will be evaluated when a new code entity is selected to replay the macro. This configuration can be done by using a Domain Specific Language (DSL). Figure 5.3 shows an example of manual configuration in PackageManager, This panel is available when the developer right-clicks on a parameter.

In this example, the developer modifies the parameter methodContent1. This parameter represents the content of the soon-to-be-added method platformRequirements(). As discussed in Section 4.2.1, this method returns a string which is extracted from the code of the soon-to-be-removed method platform(); more specifically, the argument of the invocation to addPlatformRequirement(String).



Please define the new value as a text expression (e.g., the name of a class/method). The value is only changed if the expression is valid

```
method = @class1.getMethodNamed(@method1);
packageName = null;
for node in method.getAST().getAllChildren() {
    if (node.isMethodCall() & (node.name == 'addPlatformRequirement')) {
        packageName = node.getArgumentAt(1);
        break;
    }
}
return "platformRequirements() { return { " + packageName.asString() + "} }";
```

Figure 5.3: Configuring parameters in MacroRecorder (right click on a parameter). Existing parameters, *e.g.*, @class1 and @method1, are referenced with their unique identifiers.

Using our proposed DSL, the developer first retrieves the content of the method platform, referencing the class and the method in the parameters of the pattern (first line). Next, the developer searches for the invocation to addPlatformRequirement(String), then the argument of this invocation is stored (for iteration). The result of the expression is the source code of the method platformRequirements (last line).

This DSL allows the developer to reference parameters in other transformations, using the symbol "@". Moreover, this language provides query features to inspect the source code. Basically, an expression can:

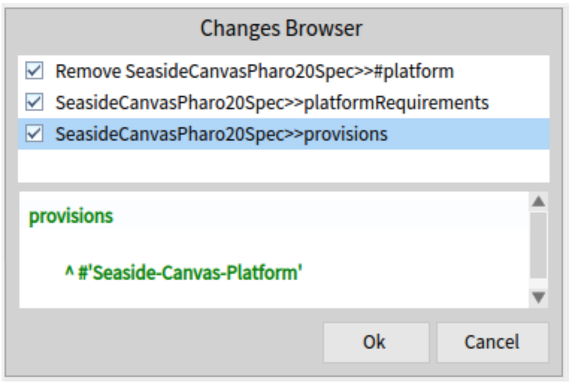
- instantiate other parameters by referencing their unique names (*e.g.*, the construct @method1 returns the value of this parameter);
- use the program querying language to evaluate a value from the parameters (*e.g.*, using the method getMethodNamed); and/or
- define a value directly (*e.g.*, @method1 = 'addPlatformRequirement').

To applying the macro to a new code entity, the developer selects a code entity on which the tool must perform the recorded transformations. The expression will be re-evaluated and it might return a different result. Consequently, the parameters will be re-computed for this new code entity.

5.2.3 Replaying Code Transformations

After pressing the "*Replay*" button (Figure 5.2.C), MACRORECORDER performs the transformation operators. If no parameter was configured, the tool will replay the transformations exactly as they were initially recorded. Moreover, if there is an execution error when performing at least one transformation operator, MACRORECORDER rolls back all the changes done before the failing operator. More specifically, replaying a transformation operator might fail because: (i) an exception was thrown during the transformation, *i.e.*, the code entity to be transformed could not be retrieved in the new location; or (ii) the transformations in the macro produced code that was not compilable.

MACRORECORDER produces a sequence of concrete transformations that will be performed by the IDE. Figure 5.4 shows the result of the transformations. This result is the normal behavior of the IDE before performing any refactoring transformation. MACRORECORDER only reuses this infrastructure. The developer can check whether the transformations are correct and accept them afterwards.



☑ Remove SeasideCanvasPharo20Spec>>#platform
☑ SeasideCanvasPharo20Spec>>platformRequirements
☑ SeasideCanvasPharo20Spec>>provisions

```
provisions

    ^ #'Seaside-Canvas-Platform'
```

Figure 5.4: Result of applying PACKAGEMANAGER's macro with MACRORECORDER.

5.3 MACRORECORDER's Architecture

In this section we present MACRORECORDER approach to record, configure, and replay transformation patterns. The current implementation of the tool is developed in Pharo.[1] Pharo is an open-source, Smalltalk-inspired, dynamically typed language and environment. The approach itself can be applied to other languages, such as Java. For each step, the approach has specific requirements, listed as follows:

- a code transformation recorder (*e.g.*, MYLYN [KM06], SYDE [HL10], CHEOPSJ [SD12], COPE [NCDJ14], or EPICEA [DBG+15]). The recorder is an extension

[1]http://pharo.org

of an IDE (*e.g.*, Eclipse, Pharo) which is responsible for monitoring code edition activities and storing code transformations as first class events;

- a program querying tool (*e.g.*, PQL [MLL05], Ekeko [RS14]). The approach must be able to inspect specific properties of the code entities (*e.g.*, check whether a method calls addPlatformRequirement()) to generalize the transformation to other code locations; and
- a code transformation tool (*e.g.*, Eclipse's refactoring tools, Refactoring in Pharo). Each transformation operator recorded during the first step must be parameterizable. For this purpose, the transformation tool must be able to represent fine-grained transformations.

Our approach is highlighted in grey in Figure 5.5. Thus, specifically for recording and replaying steps, the existing Pharo tools have been extended to fit our requirements.

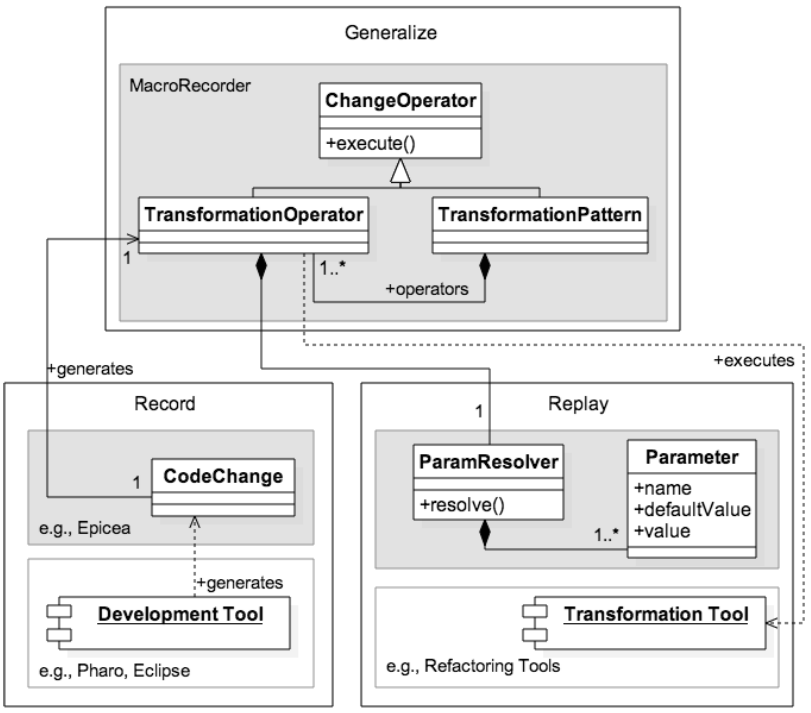


Figure 5.5: Overview of MacroRecorder's approach (highlighted in grey). The transformation operator establishes the connection between recorded code transformation events and transformation algorithms in the transformation tool. We discuss each step in Sections 5.3.1 to 5.3.3.

5.3.1 Recording Code Transformations

MacroRecorder uses Epicea [DBG+15], a tool that records developer events in the Pharo IDE, including source code transformations. MacroRecorder records both level one and level two transformations. For example, an *Add Method* transformation event also stores the content of the method, instead of recording the addition of an empty method (level one) then one operator for each added statement (level one transformation).

We summarize the transformations recorded by our tool in Figure 5.6. The events highlighted in grey were added in MacroRecorder from Epicea's original model. In total, we added 14 transformation events to the existing Epicea model, which contained 45 events. We extended class transformation events to also consider attributes (addition and removal) and inheritance modification (*e.g.*, change the superclass), summing up three new class transformation events. Similarly, we extended method code changes to consider modification in the protocol (method classification in Pharo, one event), and to also consider addition and removal of temporary variables, assignments, return statements, pragmas, and message sends. Thus we added eleven new method transformation events.

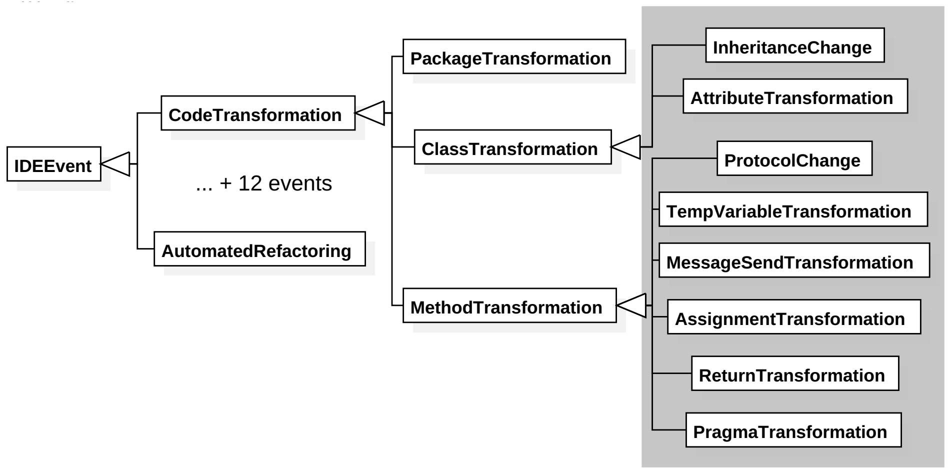


Figure 5.6: Transformation events recorded by Epicea and MacroRecorder from Pharo IDE. Events highlighted in grey were extended from the Epicea's original model. Each "*Transformation*" event is a generalization of "*Addition*" and "*Removal*" events.

5.3.2 Configuring Code Transformations

After recording with Epicea, MacroRecorder converts the resulting events in transformation operators. First, to calculate the parameters for each transformation. And second, to specify to the transformation tool how the transformation will be performed automatically. To exemplify this definition, Listing 5.1 presents MacroRecorder's extension of one particular transformation event that generates a *Remove Method* transformation operator.

Listing 5.1: Extension in Epicea's *Method Removal* transformation event.

```
asTransformation() {
 operator = new TransformationOperator("RemoveMethod");
 operator.setTransformation( RemoveMethodTransformation );
 operator.setTransformationMethod("removeMethod(Class,String)");
 operator.addParameter(this.methodRemoved().getName());
 operator.addParameter(this.methodRemoved().getClass());
 return operator;
}
```

This extension defines: (i) the method that will execute this operator in the transformation tool; and (ii) a sequence of parameters that are necessary to replay it, which are retrieved by Epicea's transformation event. More specifically, the transformation tool has a class RemoveMethodTransfomation that, when its method removeMethod(Class, String) is called with the correct arguments, will remove the method from the specified class.

The names of the parameters are unique and they are calculated automatically. If two or more transformations have identical parameters during recording, *i.e.*, the name of the class is the same in PackageManager's working example, these transformations will have a reference to a single parameter, *e.g.*, named @class1, as presented in Figure 5.2. Parameters can be also redefined as expressions by the developer (see Section 5.2.2).

5.3.3 Replaying Code Transformations

Finally, MacroRecorder executes the macro. First, the tool must obtain the parameters that each transformation operator requires. If a parameter was configured either manually or automatically (next chapter), MacroRecorder assigns this parameter to its corresponding transformation operator. Otherwise, the parameter will return the value as recorded. Ultimately, MacroRecorder executes the transformation operators in sequence. The operators are performed by the IDE. The IDE shows the result of the transformations in which the developer can check whether they are correct.

5.4 Validation Experiment

In this section, we evaluate MacroRecorder with real sequences of code transformations in software systems. We presented these sequences in Section 4.4.1. Our evaluation follows the methodology used in related work [MKM11]. We evaluate MacroRecorder's *complexity* when one must configure transformations manually (see Section 5.5.1). We evaluate the tool's *accuracy* to check whether the tool is able to record and replay the transformations (Section 5.5.2). Finally, we evaluate the *similarity* of the source code result of automatic transformations in comparison with manual edition by the developer (Section 5.5.3).

5.4.1 Research Questions

We propose research questions to discuss the ability of MacroRecorder to perform transformation patterns. The challenge in such activity consists in replaying code transformations in code locations that are similar, but non identical, to the one in which the transformation pattern was recorded. We propose a main research question:

RQ5.1 *Can we use macros to perform transformation patterns with MacroRecorder?*

Assessing MacroRecorder. To complement RQ5.1, we also evaluate MacroRecorder approach from the point of view of the developer (as a future user). Such evaluation is important to motivate the developer to use the tool instead of manually and repetitively perform code transformations. We describe the complementary research questions as follows.

RQ5.2 *How many parameters must be configured manually using MacroRecorder?* We investigate how many parameters are necessary to replay a macro in another code location.

RQ5.3 *Are macros performed accurately in each occurrence of a transformation pattern?* We investigate whether MacroRecorder is able to perform transformations in other code locations.

RQ5.4 *Is the source code result of transformations similar to the manual edition by the developer?* We investigate whether MacroRecorder generates source code that is correct according to the manual edition from the developer.

5.4.2 Target Systems

Our dataset is based on the transformation patterns we identified in Chapter 4. Because MacroRecorder is currently implemented in Pharo, we selected the systems implemented in this language for evaluation. From this selection, we had originally six instances of transformation patterns in three systems. Additionally, we released MacroRecorder for the Pharo community in November of 2015.[2] Thus

[2]http://forum.world.st/MacroRecorder-available-in-Pharo-td4865061.html

far, we received instances of transformation patterns in two new systems from the community, and an additional pattern in the PetitSQL tool. The additional systems are described as follows.

MooseQuery is a framework to query dependencies between entities in the FAMIX model [DAB+11]. It was restructured to be language independent (the original implementation focused on object-oriented languages).

Pillar is a language and family of tools to write and generate documentation in text, PDF, HTML pages, etc. The tests were restructured to provide a simpler and reusable interface. It is also built using PetitParser, among other frameworks.

Similar to our evaluation in Chapter 4, we evaluate two versions of each system, *e.g.*, before and after the systematic code transformation took place. Table 5.1 summarizes descriptive data about the systems, including the Pharo systems we selected from Chapter 4. Exceptionally for Pillar, the developers did not perform the transformation pattern manually. The developers described both (i) the transformations that should be performed, and (ii) the collection of code entities to which the macro should be applied. We used MacroRecorder to perform the transformations.

Table 5.1: Size metrics of our dataset. Each line describes a rearchitecting between two versions. Metrics are shown in pairs (before and after the rearchitecting). Pillar is the project for which the developers did not perform the transformations manually, therefore there is no version after the rearchitecting.

	Packages	**Classes**	**KLOC**
PetitDelphi 0.210 / 0.214	7 / 7	313 / 296	8 / 9
PetitSQL 0.34 / 0.35	1 / 1	2 / 2	0.3 / 0.4
PackageManager 0.58 / 0.59	2 / 2	117 / 120	2.5 / 2.3
MooseQuery 0.245 / 0.266	2 / 2	3 / 3	0.2 / 0.2
Pillar 0.178 / –	24 / –	278 / –	14 / –

5.4.3 Recording Macros with MacroRecorder (RQ5.1)

In a practical setting, MacroRecorder requires two code entities as input: (i) the recording entity, as the first code entity modified to record the macro; and (ii) the replaying entity, where the macro must be replayed by our tool. We use the source code *before* the rearchitecting effort to record one sequence of code transformations using MacroRecorder.

We enable the recording in our tool, then we manually perform the sequence of transformations that took place. For example, we perform a sequence of *Remove*

Method and two *Add Method* transformations in PACKAGEMANAGER II, our working example in this thesis. The recording entity is selected randomly from the occurrences of the pattern. This recording phase will produce one macro for each of the transformation patterns we found.

In order to replay each macro, we first need to configure them. Specifically, we randomly select one code entity that is candidate to be transformed. The replaying entity must be different than the one which was used to record the macro. We iteratively configure as many parameters as needed until MACRORECORDER automatically performs the macro in the replaying entity with success. We perform this configuration by manually defining expressions that will be evaluated when a new code entity is selected to replay the macro (see Section 5.2.2). This process is only executed once for each macro, in order to generalize it for a second occurrence. Once the macro is configured, we automatically perform it in all the occurrences, including both the recording and replaying entities.

In total, we recorded ten macros: one macro for each transformation pattern in the Pharo systems we investigated in Chapter 4 (totaling six macros), and four new ones: a second macro in PETITSQL, two in MOOSEQUERY, and one in PILLAR. Table 5.2 presents descriptive data about the macros we recorded. It describes, for each example: the number of occurrences of the pattern, the number of operators obtained after the recording stage, and the number of parameters as calculated automatically by MACRORECORDER. We reference the number of occurrences as a metric ($occurrences(M)$) in Section 5.4.4. We discuss the number of parameters we actually configured in Section 5.5.1.

Table 5.2: Descriptive metrics of macros we recorded.

Transformation patterns	Pattern occurrences	Number of operators	Number of parameters
PetitDelphi	21	2	3
PetitSQL I	6	3	5
PetitSQL II	98	3	6
PackageManager I	50	5	4
PackageManager II	19	3	5
PackageManager III	64	2	4
PackageManager IV	7	4	7
MooseQuery I	16	1	3
MooseQuery II	8	4	5
Pillar	99	4	9

5.4.4 Evaluation Metrics

In this section, we present the metrics we use in the evaluation of our approach. To evaluate the complexity of our approach, we measure how many parameters must be configured to apply a macro in another code entity.

- **Number of configured parameters** is the number of parameters the developer needs to configure in order for MacroRecorder to perform the macro in a new code entity.

We expect that only a subset of all parameters in a macro must be configured. We showed in our working example in PackageManager that the methods being systematically removed have the same signature, *e.g.*, `platform()`. The parameter that references this method does not need to be configured in this macro.

After selecting a replaying code entity, MacroRecorder reconfigures the parameters of the macro and tries to instantiate the transformations in this new location. If successful, the macro will transform the source code in this location. To check accuracy of the resulting code, we use the source code *after* the rearchitecting as our gold standard. Note that using the manual transformations as gold standard presents some difficulties as some transformation operators in MacroRecorder can be applied in different ways. We now present an example of an issue that might occur.

In MacroRecorder, the transformation operators are executed independently, *i.e.*, one operator has no knowledge of the operators that might have been performed before. We observed one case in which one transformation, namely *Remove Assignment Statement*, was performed before a *Add Assignment Statement* transformation. However, the latter did not register the position from which the statement was removed. Consequently, MacroRecorder added code in a different position in comparison with the developer. Nevertheless, the resulting source code might be different but considered semantically equivalent, *e.g.*, the code "`x := 5; y := 4`" as produced by the developer, and "`y := 4; x := 5`" as produced by the tool are equivalent.

In order to evaluate each time we replay a macro, we propose the following non-exclusive classification.

- **Matched** is an occurrence in which all the transformation operators were performed. This category relates to the ability of the approach to instantiate the transformations in a new location;
- **Correct** is an occurrence in which the resulting code is behavior-equivalent to the gold standard. We make this classification by manual code inspection. This category relates to the ability of the approach to transform code that is accurate to the developer's manual edition.

Consequently, consider a macro M with $occurrences(M)$ occurrences. Therefore, $matched(M)$ is the number of occurrences in which the macro M performed all the transformations. Similarly, $correct(P)$ measures the number of occurrences that are classified as correct. The following metrics are proposed in related work on automated code transformation [MKM11].

$$coverage(M) = \frac{matched(M)}{occurrences(M)} \tag{5.1}$$

$$accuracy(M) = \frac{correct(M)}{occurrences(M)} \tag{5.2}$$

Therefore, coverage measures the percentage of the occurrences for which MacroRecorder was able to instantiate and perform the transformations. Moreover, accuracy measures the percentage of occurrences in which the modified code is equivalent to the result of manual edition.

We additionally calculate for the correct occurrences, the similarity between the result of manual and automatic transformations. For each changed code entity, we retrieve its AST tree c. Therefore, given the results of both manual (c_{manual}) and automatic (c_{auto}) transformations, similarity is defined as:

$$similarity(c_{manual}, c_{auto}) = \frac{|(c_{manual} \cap c_{auto})|}{|(c_{manual} \cup c_{auto})|} \tag{5.3}$$

Thus, similarity is also a percentage metric. Similarly to coverage and accuracy, the similarity in a transformation pattern P calculates the average similarity to all the code entities modified in all the occurrences of this pattern.

5.5 Experiment Results

5.5.1 How many parameters must be configured manually using MacroRecorder? (RQ5.2)

In this evaluation, we investigate how many parameters are necessary to be configured so that a macro can replay the transformations in another code location. Table 5.3 describes the number of parameters we had to modify to perform each macro for all occurrences of the transformation pattern.

As expected, not all parameters needed to be configured in our macros, even Pillar's macro which has the most parameters in our dataset. Specifically, 62% of the total number of parameters, or around three parameters per macro, needed to be configured in our examples.

Although a subset of parameters must be configured, some of them might be more complex to do so. We discuss an example of expression we wrote for Pack-

Table 5.3: Number of configured parameters for each recorded macro.

Macros	Manually configured	Total Number of parameters
PetitDelphi	2	3
PetitSQL I	2	5
PetitSQL II	3	6
PackageManager I	3	4
PackageManager II	3	5
PackageManager III	2	4
PackageManager IV	5	7
MooseQuery I	2	3
MooseQuery II	4	5
Pillar	6	9
Average	3	5

ageManager in Section 5.2.2. In order to retrieve the right parameter, the expression had to iterate over all of the AST nodes in a method, look for a specific method invocation, and then specify the body of the method to be added.

We found another case in which MacroRecorder is limited in the specification of more complex macros. This limitation is also found in related work [MKM11], and we found this case in PackageManager's second macro (our running example in this thesis). We observed five occurrences in which the platform() method does not invoke the method addProvision(String). Therefore, the expected behavior would be not to add a method provisions() in these occurrences.

However, in MacroRecorder, the macros are executed as recorded, *i.e.*, all the transformation operators must be performed, even the one that adds a provisions() method. If we configured this macro as presented in Section 5.2.2, replaying it in these occurrences would raise an exception because the invocation to addProvision(String) does not exist, therefore the entire macro would not be applied. We opted to return the parameter as it was recorded in this example, and the result of transformations is shown in Listing 5.2.

In this case, the macro covered the variant occurrences of this transformation pattern. However, the resulting code is not correct according to manual edition, *i.e.*, the method provisions() should not be created. We observed that this limitation could be solved if MacroRecorder allowed the developer to add a custom precondition to a subset of transformations, *e.g.*, *Add Method* named provisions() only if there was an invocation to addProvision(String) in the method platorm().

Listing 5.2: Modified code in SeasideTestsPharoCanvasSpec. The addition of the method provisions is result from the transformation pattern as it was recorded.

```
– public void platform() {
–     package.addPlatformRequirement("squeakCommon");
– }

+ public String[] platformRequirements() {
+     return { "squeakCommon" };
+ }

+ public String[] provisions() {
+     return { "Grease–Core–Platform" };
+ }
```

Summary. Almost two-thirds (64%) of the parameters of a macro needed to be configured. However, defining their corresponding expressions might be complex for developers. The results show that, with few limitations, we were able to configure MacroRecorder to replay the macro in other locations. We discuss whether MacroRecorder also transforms the remaining occurrences of the transformation pattern in the next section.

5.5.2 Are macros performed in each occurrence of a transformation pattern? (RQ5.3)

We now investigate how the automated code transformation performed by MacroRecorderwas applied considering all of the occurrences of the transformation patterns. We measure both coverage and accuracy of the macros, as discussed in Section 5.4.4. Table 5.4 summarizes the results.

Table 5.4: Accuracy results for each transformation pattern.

Macros	Pattern occurrences	Matched	Correct	Coverage (%)	Accuracy (%)
PetitDelphi	21	21	20	100	96
PetitSQL I	6	4	4	67	67
PetitSQL II	98	98	98	100	100
PackageManager I	50	10	10	20	20
PackageManager II	19	19	14	100	74
PackageManager III	64	64	64	100	100
PackageManager IV	7	7	7	100	100
MooseQuery I	16	15	14	94	88
MooseQuery II	8	8	8	100	100
Pillar	99	99	82	100	83
Average				88	84

In general, 88% of the pattern occurrences matched. The lowest coverage result was found in PackageManager's first macro. In this system, packages are represented as data objects. The method dependencies() defines the packages on which a package depends. Similar to the second pattern in PackageManager, the developers updated the dependencies definition so that it only provides data. Listing 5.3 shows the result of the transformations in a package defining one dependency.

Listing 5.3: Modified code in the class `Seaside31Spec`.

```
public dependencies() {
–    package.addDependency("Seaside–Group–Default")
–       .addVersion(package.getVersion());
+   dependencies = new Collection();
+   dependencies.add( new Pair("Seaside–Group–Default","3.1.0") );
+   return dependencies;
}
```

We recorded a macro in a package that defined one dependency. However, other packages define more than one. Due to the fact that MacroRecorder is limited to replay the exact sequence of transformations as recorded, the tool tried to replace the first invocation to addDependency(String), adding a return statement in the middle of the method, *e.g.*, other dependencies were specified after the modified one. The resulting code could not compile, and the transformations of the macro were not effectively applied. This limitation is also found in related work [MKM11]. We observed that the coverage would increase if MacroRecorder allowed the developer to select a subset of transformations that could repeat internally in each occurrence.

For accuracy, in 84% of the occurrences, the resulting source code is behavior-equivalent to developer's manual edition. Similar to coverage results, we observed that the accuracy is low due to small variations in the source code. We discussed in the previous section one of the examples in which a subset of the transformation operators of a macro should not be performed, leading to incorrect code in comparison with the transformations performed by the developer.

These five pattern occurrences matched because all the transformations were performed, however the resulting code is not correct according to manual edition. We observed that the accuracy would increase if MacroRecorder allowed the developer to add a precondition to a subset of transformations, *e.g.*, *Add Method* if a condition applies.

Summary. The macros could be replayed in 88% of the occurrences. The result of automatic transformation with MacroRecorder is 84% correct according to manual edition by the developers. The results show that it is possible to perform transformation patterns with MacroRecorder. We identified some limitations of our approach when patterns occurred with small variations.

5.5.3 Is the source code result of transformations similar to the manual edition by the developer? (RQ5.4)

We investigate whether the source code transformed by MacroRecorder is similar in comparison with the manual edition by the developers. These results are complementary to the accuracy ones. More specifically, we measured similarity in the occurrences where MacroRecorder performed the macro correctly, *i.e.*, the 84% of correct occurrences in Section 5.5.2. Table 5.5 summarizes the results.

Table 5.5: Similarity results for each macro.

Macros	Pattern occurrences	Correct	Similarity (%)
PetitDelphi	21	20	100
PetitSQL I	6	4	85
PetitSQL II	98	98	99
PackageManager I	50	10	100
PackageManager II	19	14	87
PackageManager III	64	64	100
PackageManager IV	7	7	68
MooseQuery I	16	14	100
MooseQuery II	8	8	100
Pillar	99	82	100
Average			94

The resulting code in the correct occurrences is 94% similar to developers' manual edition. Concerning the number of correct occurrences and similarity, the macros in PetitDelphi and PackageManager (the third one) had the best results in the study. In PetitDelphi, the pattern consists in removing methods and classes from the system. Our tool covered all of these occurrences. In PackageManager, the macro creates methods with only one statement. Therefore, from the occurrences that matched the context, MacroRecorder replays them exactly like the developer.

For PackageManager's fourth macro, even though MacroRecorder produced correct transformations, the transformed code is not completely similar to the manual edition by the developer. In this case, the tool executed an *Add Assignment Statement* in a different position in the source code, in comparison with the developer. The result is shown in listings 5.4 and 5.5. In these cases, we consider the code is still correct although it is slightly different from the developer's edition.

Listing 5.4: Result of manual edition in SolverResolvedDependencyForTest. The developer basically edited the method invocation in place.

```
public testResolvedDependency() {
  // ...
  solver = new Solver().add(repository);
  dependency = new Dependency()
    .setPackage(package)
-   ;
+   .setVersion(new VersionConstraint("3.1");
  // ... }
```

Listing 5.5: Result of automatic transformation in SolverResolvedDependencyForTest, via MacroRecorder. The tool recorded the transformations as a removal then addition of a method call, which was placed in another location.

```
public testResolvedDependency() {
  // ...
+ dependency = new Dependency()
+   .setPackage(package)
+   .setVersion(new VersionConstraint("3.1");
  solver = new Solver().add(repository);
- dependency = *new* Dependency()
-   .setPackage(package);
  // ... }
```

The first modification was performed by the developer, retrieved from source code history; the second one is result of automatic transformation using MacroRecorder. In this example, the correct statement was automatically removed from code. However, to add the new statement, the transformation operator calculated a different location.

Summary. In the occurrences where MacroRecorder performed the transformations correctly, 94% of the resulting code is similar to the manual edition from the developer.

5.6 Threats to Validity

5.6.1 Construct Validity

The construct validity is related to whether the evaluation measures up to its claims. In our evaluation, we measured complexity of our approach by the number of parameters that needed to be manually configured to replay the macro (RQ5.2). However, we did not rely on real developers effectively using the tool. One may argue that even with a low number of parameters to be configured, elaborating expressions for these parameters can still be a complex task.

We acknowledge this issue. In fact, measuring quantitively the complexity of the expressions we described proved to be a difficult task. Moreover, we previously discussed that the implementation of the tool is a proof-of-concept one. The fact that only a subset of the parameters were configured is a good result. The evaluation concerned the limitations of an automated support to apply macros. We focus on the automated configuration of the parameters in Chapter 6, and we intend to evaluate the usability of the tool with real developers in future work.

5.6.2 Internal Validity

The internal validity is related to uncontrolled factors that might impact the experiment results. In our evaluation, we discussed examples in which our tool generated source code that was incorrect in comparison with manual edition from the developers (RQ5.3). During automatic code transformations with macros in a real case, we do not consider the side effects caused by introducing incorrect code, such as introducing bugs, adding unexpected behavior, *etc*.

We acknowledge this issue. However, using MacroRecorder's prototype in a practical setting, the developer must select each new occurrence, *i.e.*, code that should be transformed, instead of transforming all occurrences at once. The tool then shows the result of the automated transformations before actually changing the code (see Section 5.2.3). The developer can discard the transformations if the code is incorrect, and the system is not compromised.

5.6.3 External Validity

The external validity is related to the possibility to generalize our results. First, most of the systems in our dataset are small. This threat was discussed when we discovered transformation patterns in these systems in Chapter 4. In MacroRecorder's evaluation, four new transformation patterns were suggested by the community. Two of them, PetitSQL II and Pillar, were considerably recurrent, with 98 and 99 occurrences, respectively. This fact only reinforces that the size of the system is not an issue.

And second, Pharo is a language with a simple grammar. One might argue that the approach does not generalize for other languages and IDEs. We do not acknowledge this fact as a threat. Using this proof-of-concept tool, we were able to replay transformation patterns even in systems we did not know, *e.g.*, the Pillar tool which was suggested by the community. In our evaluation, we found similar limitations and results as related work.

5.7 Summary

In this chapter, we presented MacroRecorder, a proof-of-concept tool to record and replay sequences of code transformations, *e.g.*, macros. Using this tool, the developer records the sequence of transformations once, then generalizes the recorded macro to apply it automatically in other code locations. We discussed our approach which extends both development and transformation tools, to record code transformations that are parametrizable and able to be replayed.

We evaluated our tool using real cases of systematic code transformation, as presented in Chapter 4. Additional cases were also suggested by the Pharo community. We recorded macros for all the transformation patterns which were discovered in Pharo systems.

To generalize the macros, we had to configure an average of three out of five parameters per macro. However, this generalization phase was complex for some systems. Employing the manual configuration, MacroRecorder was able to perform 88% of the examples with 84% accuracy. The source code resulting from automatic transformation is 94% similar to manual code edition. We discussed specific features in our case study that MacroRecorder can overcome in order to improve its accuracy. The evaluation leads to the conclusion that customizable, system specific, source code transformations can be automated.

We propose to automate the application of the macro in Chapter 6. First, we propose automatic configuration of the parameters. We propose to analyze dependencies between parameters to infer abstract expressions automatically. For example, which AST nodes are similar between the code entity in which the macro was recorded and the code entity where the macro should be replayed. And second, we propose, after recording a macro and replaying it a couple of times, to recommend other locations in the source code that could also be transformed by this macro. Such automation would avoid errors of omission due to the manual reapplication of a macro, *i.e.*, this automated support would transform all occurrences of a transformation pattern at once.

6 Automating Systematic Code Transformation

Contents

6.1 Introduction

In the previous chapter, we proposed to record sequences of code transformations to later replay them in other code locations. Replaying a sequence of code transformation in different, but similar, code locations requires the developer to evaluate how the transformations should be configured in each new location. We come back to the working example in this thesis to motivate such configuration. Table 6.1 summarizes the parameters in the examples as first presented in Chapter 4. We come back to this table later in this chapter.

Table 6.1: Parameters required to perform the transformations in the classes GreasePharo30CoreSpec and SeasideCanvasPharo20Spec.

Parameter	GreasePharo30CoreSpec	SeasideCanvasPharo20Spec
Modified Class	GreasePharo30CoreSpec	SeasideCanvasPharo20Spec
Removed Method	platform()	platform()
Added Method	platformRequirements()	platformRequirements()
Added Return Statement	{ "pharo" }	{ "pharo2.x" }
Added Method	provisions()	provisions()
Added Return Statement	{ "Grease-Core-Platform" }	{ "Seaside-Canvas-Platform" }

The transformations were performed in two different classes, named GreasePharo30CoreSpec and SeasideCanvasPharo20Spec. Some parameters are similar in both examples, *e.g.*, the signatures of the methods to be removed and added in

Table 6.1, and therefore would not require configuration. However, other parameters are non-identical, *e.g.*, both return statements in Table 6.1 are different from one class to the other. Specifically in this case, they are retrieved from the source code of method `platform()`. We found other cases in which parameters derive from the name of the class, from the source code of a different class, *etc*. In Chapter 5, we proposed manual configuration from the developer to replay a macro.

This chapter presents an approach to automatically configure a macro. In a practical setting, the developer selects a different code location in which the tool must replay the macro. The tool attempts to configure the transformations in this new location, by comparing the selected location with the one in which the macro was originally recorded. The automatic configuration approach adapts the parameters accordingly. If successful, MacroRecorder replays the transformations automatically in the selected location.

From the moment that a macro is ready to be replayed, MacroRecorder requires from the developer to find all the code locations where the macro can be applied, then to replay MacroRecorder in each new location. In our working example in this thesis, one could notice that classes containing a method `platform()` were transformed. We identified such *"application condition"* for the patterns we found in Chapter 4 in Section 4.4.1. One could use this information as a hint to identify other classes that need to be transformed.

However, it might not be clear for the developer whether this simple condition is correct, necessary, and/or sufficient to find all the correct locations in the system. In other systems, the condition might be more complex than just considering the existence of a specific method. Such task involves inspecting the entire source code of the system. As presented in Chapter 4 in Section 4.4, developers forgot candidates for transformation, some of these appearing from time to time as other transformations bring them to light.

This chapter also presents an automatic recommendation approach. We evaluate three distinct code search approaches to find code locations that would require similar transformations. We validate the resulting candidate locations from these approaches on the systematic code transformation cases we study in this thesis.

The main contributions of this chapter are summarized as follows.

- We propose and evaluate an approach to automatically configure a macro after a new code location is selected.
- We propose and evaluate an approach to recommend code locations that are candidate for transformation using a macro.

Structure of the Chapter

Section 6.2 presents the approach to automatically configure the parameters of a macro. Section 6.3 presents the evaluation of the automatic configuration approach. Section 6.4 presents the approach to recommend code locations for transformation. Section 6.5 presents the evaluation of the recommendation approach. Section 6.6 concludes the chapter.

6.2 Automatically Configuring Code Transformations

In this section, we present our solution for automatic configuration of transformation patterns. The developer selects a new code location to replay the transformation pattern (Section 6.2.1). The tool tries to match the parameters as recorded with the code entities in the new location (Section 6.2.2). Then, the tool replays the recorded operators with their newly computed parameters (Section 6.2.3).

6.2.1 Code Entity Selection

In MacroRecorder, the developer explicitly indicates the class (or the method, statement, *etc.*) where the transformation pattern must be replayed. After pressing the button "*Replay*" in the tool, MacroRecorder starts to listen to click events in the IDE. By selecting a new code location, the developer explicitly specifies the starting point where the replayed transformations should take place.

In our PackageManager example, the developer selects the method platform() of the class SeasideCanvasPharo20Spec in the IDE. After this selection, MacroRecorder assumes this selected entity should match the first entity that was modified when recording the pattern. In this example, it was the method platform() that was removed from the class GreasePharo30CoreSpec. Listing 6.1 summarizes the code that will be compared by the tool to replay the macro in this new location.

Next, MacroRecorder tries to find which properties both entities have in common, and which properties differ. Here, both methods have the same signature. However, they belong to different classes and they have different source code. From this comparison, the tool tries to match the new code location and propagate properties that differ to the parameters in the transformation pattern.

6.2.2 Code Entity Matching

The previous step results in a pair of code entities: (i) the *recorded entity* that was first transformed when recording the transformation pattern, and (ii) the *selected entity* that was explicitly indicated by the developer to replay the transformation pattern. To replay the transformations in the selected entity, MacroRecorder tries to replace the parameter values in the transformation pattern according to properties that differ between recorded and selected entities.

Listing 6.1: Excerpt of modified code in PackageManager. The first method was removed when recording the macro. The second one was selected by the developer to replay the transformations. Both methods have very similar structure.

```
class GreasePharo30CoreSpec {
–   public void platform() {
–       package.addPlatformRequirement("pharo");
–       package.addProvision("Grease–Core–Platform");
–   }
}

class SeasideCanvasPharo20Spec {
    public void platform() {
        package.addPlatformRequirement("pharo2.x");
        package.addProvision("Seaside–Canvas–Platform");
    }
}
```

In this example, the matching between recorded and selected entities is straightforward because both methods have very similar ASTs. However, sometimes the information necessary to configure the pattern is not in the code that was transformed. For example, the receiver package on which addProvision(String) is called might be named differently in each class, or the argument ("*pharo*") might derive from the class name. In such cases, an automatic approach must search for common properties in the pair of entities, their classes, their class hierarchy, their source code, and so on.

In MacroRecorder, the automatic configuration of macro parameters relies on the notion of matching strategies.

Matching Strategies. A matching strategy is an algorithm that takes two code entities as input, *i.e.*, recorded and selected, and searches for a particular property in them. The result of a matching strategy algorithm is a map of properties $(property_{rec}, property_{sel})$ that are different between these respective entities.

For example, the strategy ClassInfo takes two class objects as input, and compares the full names of these classes and their superclasses. Identical properties, *e.g.*, same superclass, are discarded from the result. Names of attributes are not compared because they are not ordered, *i.e.*, it is not possible to perform a one-to-one match between two unordered collections of attributes. In our example in Listing 6.1, this strategy returns the pair of properties ("*GreasePharo30CoreSpec*", "*SeasideCanvasPharo20Spec*").

We now briefly describe the strategies available in MacroRecorder.

ClassInfo compares the full name of two classes and their superclasses.

ClassTokens extends ClassInfo strategy, however it compares tokens in the name of two classes (see example in Table 6.2).

ClassMethods searches, in two classes, for pairs of methods having the same signature, then compares these pairs separately using all the method strategies described below.

MethodName compares the full name of a method, similar to ClassInfo strategy.

MethodNodes searches for similar syntactic subtrees in the AST's of two methods. This strategy is a generalization and extension of the approach proposed by Meng *et al.* that retrieves the longest common subsequence of AST nodes between two methods [MKM13]. Matching nodes (*e.g.*, ArgumentNode) with different values, *e.g.*, "*pharo*" and "*pharo2.x*" in Listing 6.1 are mapped with this strategy.

MethodTokens extends MethodNodes strategy, by comparing similar tokens in pairs of AST nodes (see example in Table 6.2).

MethodInSuperclass checks whether a method M_1 is overriding another method M_2 in the superclass; if so, the strategy compares the pair of methods M_1 and M_2, using all the method strategies.

MethodCalls checks whether two methods invoke other methods in common, then it compares the full names of the receiver and the collection of arguments in both invocations.

Concrete Example. Table 6.2 describes the strategies that were applicable in the PackageManager example. MacroRecorder executes all the matching strategies discussed above by default. Each resulting mapping represents a property found in the recorded entity (*e.g.*, GreasePharo30CoreSpec.platform()) and a property found in the selected entity (*e.g.*, SeasideCanvasPharo20Spec.platform()). It is worth noting that the matching strategies look for properties by analyzing the source code. At this point, the strategies do not know the transformations parameters as they were recorded.

The matching results in Table 6.2 are candidate substitutions for this example. We refer to Table 6.1 in the beginning of this chapter, in which the parameters are similar to the candidate substitutions that the matching strategies found, as presented in Table 6.2.

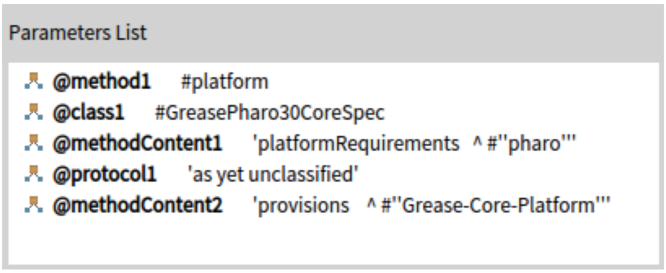
The strategy ClassName matched the names of the classes (Table 6.2, line 1), and the strategy ClassTokens only considered a fragment in their names which is different, *i.e.*, removing the token Spec (line 2). Note that these candidate substitutions produce the same result. Also, the strategy MethodNodes matched arguments in the two invocations inside the method platform() (lines 3 and 4). And finally, the strategy MethodTokens matched substrings in the names of these nodes (line 5). Again, note that the substitutions in lines 4 and 5 produce the same result. We discuss the implications of this behavior in the next section.

Table 6.2: Matching Strategies results in PACKAGEMANAGER example. Columns $property_{rec}$ and $property_{sel}$ describe matching properties between recorded and selected methods, respectively.

Strategy	$property_{rec}$	$property_{sel}$
ClassInfo	GreasePharo30CoreSpec	SeasideCanvasPharo20Spec
ClassTokens	GreasePharo30Core	SeasideCanvasPharo20
MethodNodes	"*pharo*"	"*pharo2.x*"
MethodNodes	"*Grease-Core-Platform*"	"*Seaside-Canvas-Platform*"
MethodTokens	Grease-Core	Seaside-Canvas

6.2.3 Configuring the Parameters of a Macro

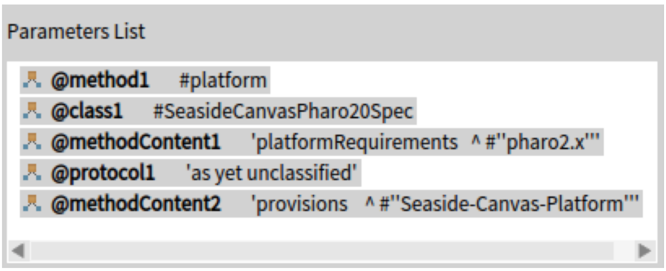
In this final step of automatic configuration, MACRORECORDER checks whether a candidate substitution can be applied in the recorded parameters. If an exact textual match is found, the tool replaces the value of the parameter automatically. Furthermore, the tool also tries to find partial substitutions. For example, the string "*pharo*" in the parameter @methodContent1 matches a candidate substitution (Table 6.2, line 3), Then, this parameter fragment is modified to "*pharo2.x*" accordingly. Figure 6.1 shows the result of the automatic configuration.



Parameters List

```
@method1   #platform
@class1   #GreasePharo30CoreSpec
@methodContent1   'platformRequirements  ^#''pharo'''
@protocol1   'as yet unclassified'
@methodContent2   'provisions  ^#''Grease-Core-Platform'''
```

(a) List of Parameters as recorded, as first shown in Figure 5.2.



Parameters List

```
@method1   #platform
@class1   #SeasideCanvasPharo20Spec
@methodContent1   'platformRequirements  ^#''pharo2.x'''
@protocol1   'as yet unclassified'
@methodContent2   'provisions  ^#''Seaside-Canvas-Platform'''
```

(b) List of Parameters after selecting class SeasideCanvasPharo20Spec to replay the macro.

Figure 6.1: Recorded (left) and reconfigured (right) parameters. All substitutions available for this example are listed in Table 6.2.

MacroRecorder also found two substitutions for the same parameter. Both strategies ClassInfo and ClassTokens (see Table 6.2, lines 1 and 2) can substitute the name of the class to be transformed, *e.g.*, GreasePharo30CoreSpec. The same behavior occurred with MethodNodes and MethodTokens strategies. In both cases, the substitutions produce the same reconfigured value, *i.e.*, substituting SeasideCanvasPharo20Spec or SeasideCanvasPharo20 in the name of the class will produce the same value: SeasideCanvasPharo20Spec. When there are multiple substitutions producing the same reconfigured value, MacroRecorder reconfigures these parameters automatically. On the other hand, if multiple substitutions produce different configuration, the tool allows the developer to choose which one is correct. We discuss some configuration issues in Section 6.3.4.

6.3 Validating Automatic Configuration

In this section, we describe the evaluation of MacroRecorder with real cases of sequences of code transformations. In Chapter 5, the macros we evaluated were configured manually. In this thesis, we focus on the automatic configuration using matching strategies.

6.3.1 Research Questions

We propose research questions to discuss MacroRecorder's ability to automatically configure macros. The challenge in such activity consists in identifying matching properties in code locations that are similar, but non identical, to the one in which the macro was recorded. In this evaluation, we focus on the automatic configuration of parameters. We already evaluated the result of transformations, *i.e.*, after replaying a macro, in Chapter 5 in Section 5.4.

RQ6.1 *How many parameters are configured automatically by MacroRecorder?* We investigate how many parameters from the transformation patterns could be configured using our proposed automatic configuration approach.

RQ6.2 *Are there parameters more likely to be configured by MacroRecorder?* We investigate whether the type of the parameter, *e.g.*, method signature, class name, or a segment of source code, has impact on the fact that the parameter will be configured by MacroRecorder. The evidence of such association might indicate for which parameter types our matching could be disabled.

RQ6.3 *How many parameters are configured correctly by MacroRecorder?* We investigate whether the matching strategies available in MacroRecorder provided *correct* configuration in our examples.

RQ6.4 *Are there parameters more likely to be configured correctly by MacroRecorder?* We investigate whether the type of the parameter has impact on the fact that the parameter will be configured correctly by MacroRecorder. The evidence of such association might indicate which parameters are more likely to require manual configuration by the developer.

RQ6.5 *Are there matching strategies more likely to provide correct configuration?* We investigate whether the matching strategies proposed in our approach have impact on the fact that parameters are configured correctly. The evidence of such association might indicate which matching strategies could be disabled for automatic configuration.

6.3.2 Recording and Replaying Macros

Our dataset consists of the macros we recorded in Chapter 5. We refer the reader back to Table 5.2 (on page 64) which describes the number of occurrences of the macro, the number of operators obtained after the recording stage, and the number of parameters as calculated automatically by MacroRecorder.

Our methodology for recording and replaying transformation patterns is similar to the one in Chapter 5. The main differences are: (i) we now perform automatic configuration, to minimize the number of parameters that need to be configured manually, and (ii) we perform all the permutations for recording and replaying transformation patterns, as opposed to selecting one random occurrence for recording and replaying on the remaining occurrences as presented in Chapter 5. We describe the evaluation as follows.

Recording Macros. Concretely, we use the source code *before* the rearchitecting effort (see Table 5.1 on page 63) to record the transformations. We select one occurrence of the pattern as the one from which the macro will be recorded. For example, a sequence of *Remove Method* and two *Add Method* transformations are performed to record a macro in PackageManager II.[1]

Replaying Macros. We select a second occurrence that will be the input for the automatic configuration approach. All matching strategies are enabled for automatic configuration. As result, the candidate substitutions are collected for further analysis.

Measure and Compare Results. We investigate whether the parameters of the macro would be configured using our approach. We also investigate whether the candidate substitutions are correct in the replayed occurrence. We present the automatic configuration evaluation in the next section.

[1]Note that three other macros were found in PackageManager.

Repeat for all pairs of occurrences. We repeat the steps discussed above for all the occurrences of the transformation pattern, including the one that was used to record the macro. For example, we record 19 distinct macros in PackageManager II. Then, we execute each macro to all the 19 occurrences, totalizing ($19 \times 19 =$) 361 permutations of record-and-replay executions for this pattern.

Note that one macro should be recorded for each occurrence. We automate the macro recording step by programming the transformations that will be recorded by the tool. The transformations might be complex, however we already have knowledge, from our evaluation in Chapter 5, which and where the macro transformations should be applied.

6.3.3 Evaluating Automatic Configuration

As discussed in Chapter 5 in Section 5.5.1, only a subset (64%) of the parameters need to be configured to apply the macro in another location. Additionally, our evaluation in Chapter 5 granted us knowledge of the right parameter configuration for each macro occurrence, because we already performed MacroRecorder on those occurrences. After performing our automatic configuration approach, our matching strategies propose a collection (sometimes empty) of candidate substitutions for each macro parameter. The candidate substitutions however might be incorrect in comparison with the right configuration. We categorize the parameters, after selecting an occurrence for replay, as follows.

- **Unchanged (UNCH)** means that the parameter should not be changed and no matching strategy found a candidate substitution for this parameter. Therefore, the parameter value remains correct.
- **Changed (CHAN)** means that the parameter was changed, whether the substitution was correct or not (see other categories later in this section).

This categorization is complementary. For each parameter, the sum of the occurrences where the parameter was *unchanged* and the occurrences where the parameter was *changed* equals to the total number of executions, *e.g.*, 361 in PackageManager II.

Evaluation for RQ6.1 and RQ6.2. We use the categorization above for RQ6.1 to investigate how many parameters were configured using our approach. Specially for RQ6.2 we also want to know whether the type of the parameter does have impact to the fact that the parameter will be configured. We elaborate the following hypotheses.

$H_0^{RQ6.2}$ Parameter type does not influence whether a parameter will be changed.

$H_a^{RQ6.2}$ Parameter type influences whether a parameter will be changed.

The *independent variable* is the parameter type, which is categorical and in our evaluation it can take six values:

- class: represents the name of a class;
- method: represents the signature of a method;
- pragma: special construct on Pharo, which is similar to method annotation: it contains signature and a collection of arguments;
- protocol: method classification in Pharo, it is represented by a short text;
- sourceCode: represents a segment of source code, *e.g.*, the value in an assignment, or the expression of a return statement; and
- variable: represents the name of a variable.

The *dependent variable* is the parameter configuration as computed by MacroRecorder. It is also categorical and it can take two values: *unchanged* and *changed*. The subjects for these experiments will be the parameter configurations, considering all the combinations of record-and-replay in our dataset.

Concerning whether the configuration proposed by our approach is correct, we propose the following categorization:

- **Incorrect Matching (INCO)** means that: (i) the parameter should *not* be changed, however one (or more) candidate substitutions were proposed for this parameter; (i) the parameter should be changed, however matching strategy was able to find a candidate substitution for it; or (ii) the parameter should be changed, one (or more) candidate substitutions were proposed, however all of them would generate an incorrect parameter value, if configured automatically.
- **Correct (CORR)** means that the parameter was configured correctly, and it equals to the total number of executions minus the ones classified as *Incorrect Matching*. A correct configuration might appear on three ways.
  - **Unchanged (UNCH)** as previously described, because it means that the parameter should not be changed and the approach did not find a candidate substitution for this parameter.
  - **Automatically Configured (AUTO)** means that the parameter should be changed and one (or more) matching strategies found a correct candidate substitution for this parameter. In the case where multiple matching strategies found candidate substitutions, we categorize the parameter as automatically configured when all the substitutions produce the same reconfigured value (see Table 6.2 on page 80).
  - **Manually Selected (MANU)** means that the parameter should be changed, multiple matching strategies found substitutions that produce different reconfigured values, and only one of these substitutions

is correct. In a practical setting, MacroRecorder allows the developer to choose which substitution is the correct one.

This categorization is also complementary. For each parameter, the sum of the occurrences where the parameter was *incorrect* and the occurrences where the parameter was *correct* equals to the total number of executions, *e.g.*, 361 in PackageManager II.

Evaluation for RQ6.3 and RQ6.4. We use the categorization above for RQ6.3 to investigate how many parameters were configured correctly using our approach. Specially for RQ6.4 we also want to know whether the type of the parameter does have impact to the fact that the parameter will be configured correctly. We elaborate the following hypotheses.

$H_0^{RQ6.4}$ Parameter type does not influence whether a parameter will be configured correctly.

$H_a^{RQ6.4}$ Parameter type influences whether a parameter will be configured correctly.

The *independent variable* is the parameter type, which is categorical and we previously described to evaluate RQ6.1. The *dependent variable* is the parameter configuration as computed by MacroRecorder. It is also categorical and it can take two values: *incorrect* and *correct*.

Evaluation for RQ6.5. Specially for RQ6.5 we want to know whether the matching strategies have impact to the fact that the parameter will be configured correctly. We elaborate the following hypotheses.

$H_0^{RQ6.5}$ Matching strategy does not influence whether a parameter will be configured correctly.

$H_a^{RQ6.5}$ Matching strategy influences whether a parameter will be configured correctly.

The *independent variable* is the matching strategy as proposed in our approach. We refer the reader to Section 6.2.2 for a short description on these strategies. The *dependent variable* is the parameter configuration as computed by MacroRecorder. It is categorical and it can take two values: *incorrect* and *correct*.

Experiment Design. To test the hypotheses we use the Chi-squared test, which can be used when there are two categorical variables. A resulting p-value lower than the 0.05 significance level means that we can reject the null hypothesis, *i.e.*, we can conclude that there is an association between the parameter type (or matching strategy) and the fact that it was changed (or correctly configured) by MacroRecorder.

6.3.4 Experiment Results

This section presents our experiments to evaluate the automatic configuration performed by MacroRecorder.

6.3.5 How many parameters are configured automatically by MacroRecorder? (RQ6.1)

In this evaluation, we investigate how many parameters from the transformation patterns can be automatically configured using our proposed configuration approach. Table 6.3 summarizes the configuration results in our dataset. The categories for each parameter were discussed in Section 6.3.3. For this research question, we focus on the parameters that were *changed* (column CHAN in Table 6.3). We discuss later in Section 6.3.7 whether the configuration was correct.

As expected, only a subset of the total number of parameters actually changed. Specifically, the parameters were changed in 59% of the total number of executions. This result is lower than the one we observed in Chapter 5 (64%), which was obtained from one execution for each macro.

We observe few parameters were unchanged in the majority of executions (*i.e.*, CHAN $\approx 0\%$ in Table 6.3). In our working example in this thesis, the method to be removed in PackageManager II is always the same (CHAN $= 0\%$ for parameter `@method1` in Table 6.3). These results indicate common properties between the most of the occurrences of the macro. Seven out of ten macros have at least one parameter with such behavior.

Summary. Considering all combinations of executions with MacroRecorder, 59% of the parameters were changed by our automatic configuration approach. In seven out of ten macros, at least one parameter per macro was unchanged in the majority of executions.

6.3.6 Are there parameters more likely to be configured by MacroRecorder? (RQ6.2)

Table 6.4 aggregates the type of parameter and the number of occurrences in which these parameters were changed. The parameters that were mostly *unchanged* were (i) class name, in 57% of the executions in Table 6.4, and (ii) protocols, *i.e.*, method classification in Pharo, in 99% of the executions. In PetitDelphi and PetitSQL II, the class (`@class1`) from which the method (`@method1`) is removed is always the same (CHAN $= 0\%$ for this parameter in Table 6.3).

We applied Chi-square to test if there is an association between parameter type and the fact that it was changed. According to the test (p-value = $2.2\mathrm{e}{-16}$), we

Table 6.3: Automatic configuration results. See text for column descriptions.

Macro	Parameter	UNCH	AUTO	MANU	INCO	ALL	CHAN	CORR
PetitDelphi	@method1	21	417	3	0	441	95.24%	100.0%
	@class1	441	0	0	0	441	0.00%	100.0%
	@class2	40	381	1	19	441	90.93%	95.7%
PetitSQL I	@method1	5	19	9	3	36	86.11%	91.7%
	@selector1	28	0	0	8	36	22.22%	77.8%
	@source1	4	0	0	32	36	88.89%	11.1%
	@protocol1	28	0	0	8	36	22.22%	77.8%
	@class1	28	0	0	8	36	22.22%	77.8%
PetitSQL II	@method1	98	9506	0	0	9604	98.98%	100.0%
	@source1	98	2	0	9504	9604	98.98%	1.0%
	@class2	9604	0	0	0	9604	0.00%	100.0%
	@protocol1	9507	0	0	97	9604	1.01%	99.0%
	@variable1	98	9506	0	0	9604	98.98%	100.0%
	@class1	9604	0	0	0	9604	0.00%	100.0%
PackageManager I	@return1	1	0	0	499	500	99.80%	0.2%
	@class1	1	39	10	450	500	99.80%	10.0%
	@selector1	1	39	0	460	500	99.80%	8.0%
	@method1	499	0	0	1	500	0.20%	99.8%
PackageManager II	@class1	1	18	0	342	361	99.72%	5.3%
	@protocol1	361	0	0	0	361	0.00%	100.0%
	@source2	0	0	0	266	266	100.00%	0.0%
	@method1	361	0	0	0	361	0.00%	100.0%
	@source1	144	0	84	133	361	60.11%	63.2%
PackageManager III	@value1	3115	256	143	582	4096	23.95%	85.8%
	@class1	1	30	33	4032	4096	99.98%	1.6%
	@selector1	4093	0	0	3	4096	0.07%	99.9%
	@method1	4054	0	0	42	4096	1.03%	99.0%
PackageManager IV	@class1	13	18	3	15	49	73.47%	69.4%
	@value2	15	0	1	33	49	69.39%	32.7%
	@selector2	49	0	0	0	49	0.00%	100.0%
	@method1	9	40	0	0	49	81.63%	100.0%
	@variable1	25	16	0	8	49	48.98%	83.7%
	@value1	16	6	7	20	49	67.35%	59.2%
	@selector1	49	0	0	0	49	0.00%	100.0%
MooseQuery I	@pragma1	12	20	21	172	225	94.67%	23.6%
	@class1	1	0	14	210	225	99.56%	6.7%
	@method1	177	0	0	48	225	21.33%	78.7%
MooseQuery II	@method1	49	50	0	1	100	51.00%	99.0%
	@pragma1	2	0	0	98	100	98.00%	2.0%
	@method2	2	14	4	80	100	98.00%	20.0%
	@pragma2	2	0	0	98	100	98.00%	2.0%
	@class1	4	12	4	80	100	96.00%	20.0%
Pillar	@method1	261	9050	488	2	9801	97.34%	100.0%
	@source3	12	0	1	9788	9801	99.88%	0.1%
	@source1	2	0	51	9748	9801	99.98%	0.5%
	@class2	210	2811	3285	3495	9801	97.86%	64.3%
	@protocol1	9761	0	0	40	9801	0.41%	99.6%
	@class4	7133	0	0	2668	9801	27.22%	72.8%
	@source2	12	0	1	9788	9801	99.88%	0.1%
	@class3	9565	0	0	236	9801	2.41%	97.6%
	@class1	211	2794	3567	3229	9801	97.85%	67.1%

reject the null hypothesis, *i.e.*, we can conclude that there is an association between the parameter type and whether this parameter is changed or not.

Table 6.4: Aggregated results per parameter type. @class also includes @super in Table 6.3. Similarly, @method includes @selector, *i.e.*, equivalent of method signature in Pharo. And @sourceCode includes @source, @value, and @return.

Parameter Type	UNCHANGED	CHANGED
@class	36875	27884
@method	9756	20287
@pragma	16	409
@protocol	19705	146
@sourceCode	3419	40945
@variable	123	9530

Summary. There is an association between the type of parameter and the fact that it is changed by our automatic approach. An evidence of such association indicates that parameters such as @protocol might be disabled for automatic configuration when replaying a macro.

6.3.7 How many parameters are configured correctly by MacroRecorder? (RQ6.3)

In this evaluation, we investigate whether the matching strategies were able to provide correct configuration when replaying a transformation pattern. Therefore, for this research question, we focus on the parameters that are *correct* (column CORR in Table 6.3). In average, the parameters were configured correctly in 60% of the total number of executions.

In our study, the matching strategies available in our tool were not capable of fully supporting the configuration of the macros. In our working example, the source code of method provisions() in PackageManager II was configured incorrectly in all of the executions of this macro (*i.e.*, CORR $= 0\%$ for parameter @source2 in Table 6.3). Seven out of ten macros have at least one parameter with such behavior (*i.e.*, CORR $\approx 0\%$ in Table 6.3). In these cases, and in 40% of the total number of executions, some manual configuration is necessary.

Summary. Considering all combinations of executions with MacroRecorder, 60% of the parameters were configured correctly by our automatic approach. We observed an overestimation of the matching results, meaning that our matching strategies provided incorrect matchings in some examples.

6.3.8 Are there parameters more likely to be configured correctly by MacroRecorder? (RQ6.4)

Table 6.5 aggregates parameter type and the number of occurrences in which its configuration was correct. We observe that, most of the times, parameters representing source code, *i.e.*, the content of a method, a return statement, or the value of an assignment, were configured incorrectly. For example, in PetitSQL II, this parameter type was correct in only 1% of the executions. Considering all the macros, this parameter was incorrect in 92% of the executions.

On the other hand, the parameters that were configured correctly most of the times were: variable name, in 99% of the executions in Table 6.5, protocol (99%) and method signature (98%). This fact seems to indicate with which parameter types our matching strategies are more likely to succeed and, consequently, which types of parameters are more likely to require manual configuration.

Table 6.5: Aggregated accuracy results per parameter type. @class also includes @super in Table 6.3. Similarly, @method includes @selector, *i.e.*, equivalent of method signature in Pharo. And @sourceCode includes @source, @value, and @return.

Parameter Type	CORRECT	INCORRECT
@class	49895	14864
@method	29395	648
@pragma	57	368
@protocol	19705	146
@sourceCode	3971	40393
@variable	9645	8

We applied Chi-square to test whether there is an association between the parameter type and the fact that it was configured correctly. According to the test (p-value = $2.2\mathrm{e}{-16}$), we reject the null hypothesis, *i.e.*, we can conclude that there is an association between the parameter type and whether this parameter is correctly configured or not.

Summary. There is an association between the type of the parameter and the likelihood of the parameter to be configured correctly. An evidence of such association means that some parameters, such as @sourceCode and @pragma, are more likely to require manual configuration by the developer. For such parameters, it will be required for the developer, at each time the macro is replayed, to inspect the candidate substitutions for all parameters and resolve which one is correct.

6.3.9 Are there matching strategies more likely to provide correct configuration? (RQ6.5)

Table 6.6 aggregates matching strategies and the number of occurrences in which its configuration was correct. These results are considerably higher than the ones in Table 6.5 because multiple matching strategies might suggest a candidate substitution for the same parameter. Table 6.6 then reports the categorization (*correct* or *incorrect*) for each candidate substitution separately.

We observe that MethodCalls strategy noticeably proposes more incorrect results, *i.e.*, 228702 incorrect and 74089 correct substitutions (two times more). Among the strategies that propose more correct substitutions, MethodName and ClassTokens stand out.

Table 6.6: Aggregated accuracy results per matching strategy.

Matching Strategy	CORRECT	INCORRECT
ClassInfo	80201	88757
ClassTokens	86150	83303
ClassMethods	103703	183725
MethodName	103928	65030
MethodNodes	75543	98412
MethodTokens	84565	104447
MethodInSuper	75415	94561
MethodCalls	74089	228702

We applied Chi-square to test whether there is an association between the matching strategy and the fact that it proposes correct configuration. According to the test (p-value $= 2.2\text{e}{-16}$), we reject the null hypothesis, *i.e.*, we can conclude that there is an association between the parameter type and whether this parameter is correctly configured or not.

Summary. There is an association between the type of the parameter and the likelihood of the proposed substitutions to be correct. An evidence of such association means that some matching strategies such as MethodCalls could be disabled for automatic configuration.

6.3.10 Threats to Validity

Considering the automatic configuration approach is an extension of MacroRecorder, we refer the reader to threats to validity that we discussed in Chapter 5. Those discussions still apply to this approach because focused on the configuration of the macros.

External Validity. With this automatic approach, we specifically alleviated one aspect of External Validity. In our previous evaluation in Chapter 5 in Section 5.4, we randomly selected one occurrence of the transformation pattern to record the macro. We alleviated this threat by evaluating all combinations of recording and replaying a macro, *i.e.*, considering all occurrences of the transformation pattern. In a practical setting, MacroRecorder shows the candidate substitutions to the developer after he/she selects a code entity to replay the macro. The tool then requires the developer to discard incorrect substitutions, if existing, and therefore manually configure the parameters necessary to replay the macro.

6.4 Recommending Code Locations for Systematic Transformation

In this section, we present our solution to automatically recommend code locations that are candidate to be transformed by a macro (Section 6.4.1). We use three code search approaches, using basic concepts from the literature, to find similar code locations that would require similar transformations (Section 6.4.2). We evaluate these approaches on the macros we recorded in Chapter 5 (Section 6.5).

6.4.1 Our Recommendation Approach in a Nutshell

To recommend source code locations that are likely candidates to apply a macro, one needs examples of such locations, given by the developers, from which other similar code entities can be retrieved.

Definition 9 A *code example* is a location in the source code where the macro was successfully applied.

As an example, the macro on PackageManager II starts by removing the platform() method to the class GreasePharo30CoreSpec. The first code example is the one where the macro was created, *i.e.*, recording the macro supposes that the developer successfully performed the transformations in this code location. This first code example might give initial data on the properties necessary to replay the macro in other locations.

Definition 10 A *candidate location* is an entity in the source code that is candidate to be a *code example*. Candidate location can be incorrect in two senses: (i) the macro cannot be replayed on it; or (ii) the macro could be replayed, but the developer does not wish to do so because it does not meet the, possibly informal, application conditions.

To find candidate locations for a given macro, we start from the assumption that similar code entities might be transformed in a similar way. Clone detection and code search tools can be used to identify similar code entities. We discuss some of these approaches in Chapter 3 in Section 3.4.2. In general, these tools use as input the source code of the system and one source code example. Figure 6.2 (upper part) depicts the expected behavior of such code search tools.

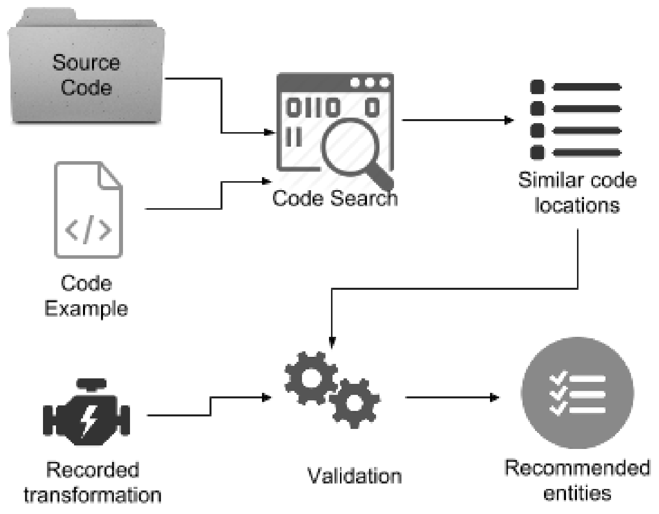


Figure 6.2: Searching code with code. Our approach retrieves code entities from an example and refine the results based on a recorded macro.

As a result, code search tools generate a list of code locations that are similar to the given example. However, for each candidate, they still require the developers to manually: (i) check whether the candidate is a correct recommendation and, if so, (ii) to effectively transform the code, *i.e.*, apply the macro. To avoid incorrect candidate locations, we propose to validate each candidate by effectively trying to apply the macro, see Figure 6.2 (lower part).

6.4.2 Approaches to Recommend Code Locations

In this section, we present our approaches to find code locations that are candidates for systematic transformation. Concretely, our approach has specific requirements:

- the source code of the entire system must be available. As a starting point, all entities in the entire source code are candidate locations;
- a macro has been created;
- one or more code examples have been specified. The code entity on which the macro was recorded already counts as one example.

The code search approaches we use are inspired by approaches in the literature. First, we search for code in similar locations, *e.g.*, same package, same superclass, etc. (Section 6.4.2.1). Second, we search for code with similar structure, as represented by their ASTs (Section 6.4.2.2). And third, we search for code with similar identifiers and comments (Section 6.4.2.3). Additionally, we use the macro to refine the list of candidate locations by checking whether the transformations can be performed on them. This process might be considered as a fourth approach (Section 6.4.2.4).

6.4.2.1 Structural approach

Nguyen *et al.* [NNP+10] identified recurring bug fixes in the code history of five real open-source systems. The recurring fixes often occurred in code locations with similar properties, such as methods containing code clones, classes extending the same superclass or implementing the same interface, methods overriding the same parent method, or classes implementing the same design pattern.

Based on these findings, we implemented a location code search approach which depends on *two* or more code examples. We call this approach "*Structural*" because it considers basic information of where the code is located. We use working example from PackageManager II to show how the approach works.

In this case, developers modified two methods with similar basic properties, as shown in Table 6.7. Both methods belong to classes in the same package (line "*package*") and inheriting from the same superclass (line "*superclass*").

Table 6.7: Properties from examples in PackageManager II. Properties are extracted from the signature of the method itself, the name of the class, superclass, and the name of the package.

	GreasePharo30CoreSpec .platform()	SeasideCanvasPharo20Spec .platform()
package	TestResources	TestResources
superclass	PackageSpec	PackageSpec
class	GreasePharo30CoreSpec	SeasideCanvasPharo20Spec
method	platform()	platform()

The structural approach then searches other entities in the system sharing the same similar properties. In the example presented in Table 6.7, the approach searches for methods with signature platform() in subclasses of PackageSpec located at the package TestResources.

The search is an all-inclusive one, *i.e.*, it assumes that all classes (or methods) in the same location, whether physical (*e.g.*, package) or logical (*e.g.*, superclass), require similar transformations. In the worst case, *i.e.*, if no similar properties (package, superclass, *etc.*) are found, the approach will recommend the original candidate set, *e.g.*, all the code entities in the entire system.

6.4.2.2 AST-based approach

Some tools have been proposed to analyze code examples to find candidates for transformations. These tools, namely LASE [MKM13] and CRITICS [ZSPK15], look for methods that have similar statements in comparison with two or more code examples. In some sense, both tools look for instances of clones, relying on the AST of methods under analysis. Based on this idea, we implemented a code search approach which depends on a single code example (as opposed to the prior work that required two).[2]

In the concrete example with PACKAGEMANAGER II, we compare the method platform() which was removed from the class GreasePharo30CoreSpec as the first transformation performed to record the macro. Assuming we want to know whether the method platform() in the class SeasideCanvasPharo20Spec is a good candidate to replay the macro, we would try to match the ASTs of both platform() methods. Listing 6.2 summarizes the code that will be compared by the tool. This example is the same we presented in Section 6.2.1.

Listing 6.2: Excerpt of modified code in PACKAGEMANAGER. The first method was removed when recording the macro. The second method will be compared to the first one to check whether it can be a candidate location.

```
class GreasePharo30CoreSpec {
-    public void platform() {
-        package.addPlatformRequirement("pharo");
-        package.addProvision("Grease-Core-Platform");
-    }
}

class SeasideCanvasPharo20Spec {
    public void platform() {
        package.addPlatformRequirement("pharo2.x");
        package.addProvision("Seaside-Canvas-Platform");
    }
}
```

First, we use a greedy text-based algorithm to compute the longest common subsequence (LCS) [Mye86] of code. When two methods have different source code, the LCS algorithm aligns what is the most common code between them. In this case, both methods have the same access to an attribute named package, and the same invocation to method addPlatformRequirement(String). Therefore, the longest common subsequence in this case is *"package.addPlatformRequirement("pharo"*. Note that the subsequence does not

[2]This algorithm was inspired by a similar one in LASE [MKM13].

include the entire statement, because the arguments are different between the two methods under analysis.

The approach then retrieves the sequence of nodes, in the AST of both methods, that contains this subsequence. It is worth noting that the LCS algorithm is used only to retrieve the most similar code and, consequently the sequence of nodes that contains this code. From there on, we compare each node of the sequence separately. In this example, the computed sequence of nodes comprises:

- the invocation to method addPlatformRequirement(String), which comprises
- the access to variable package as the receiver of this invocation, and
- the declaration of a string value ("pharo2.x").

The method in SeasideCanvasPharo20Spec shares *three* nodes with the one in GreasePharo30CoreSpec. Note that the invocation to addProvisions(String) is also common between the two methods. However, only the subtree containing the longest subsequence of code is considered for analysis.

This result is used to rank the candidate set, *i.e.*, to determine which locations are more similar to the example. The top ranking locations are then considered *candidate locations*. Table 6.8 shows the top ranked entities in comparison with method platform() in GreasePharo30CoreSpec.

Table 6.8: Top-10 most similar methods to GreasePharo30CoreSpec.platform().

Method signature	Similarity (#nodes)
SeasidePharoFlowSpec.platform()	3
SeasidePharoToolsSpecSpec.platform()	3
SeasidePharoDevelopmentSpec.platform()	3
SeasidePharoEnvironmentSpec.platform()	3
SeasidePharoWelcomeSpec.platform()	3
SeasidePharoContinuationSpec.platform()	3
SeasidePharoEmailSpec.platform()	3
SeasidePharo20ToolsWebSpec.platform()	3
SeasidePharo20CoreSpec.platform()	3
SeasideCanvasPharo20Spec.platform()	3

6.4.2.3 IR-based Approach

Information retrieval (IR) techniques use lexical analysis to search documents relevant to a query (the best known example would be the Google search engine). One of the most widely used searching model is called bag-of-words. Under this model, documents (in our case, source code text) are represented as unordered sets of terms. Then, given a query, which is also a set of terms, the IR engine retrieves documents that contain similar terms. To account for the relative importance of a term in all documents of the corpus and in each individual document, a reasonable similarity function is the *cosine similarity* of term frequency and inverse document frequency, known as TF-IDF [BR11].

We implemented a search engine which indexes source code. This approach views methods (or classes) as documents and terms are retrieved from identifiers and comments. We process each term to (i) split identifiers with the camel case and underscore naming convention; (ii) remove affixes and suffixes, (ii) discard common words that do no add meaning (stop-words); and (iii) discard words that are keywords from the programming language (additional stop-words). Table 6.9 shows set of terms extracted from SeasideCanvasPharo20Spec, where the term "*requir*" is the result of processing the original term "*requirement*".

Table 6.9: Set of terms extracted from method platform() in SeasideCanvasPharo20Spec. Terms such as "*spec*" were extracted from the name of the class.

pharo spec seasid canva
provis platform requir

This approach works with a single code example as the previous one. This code example is processed and provided to the search engine as a query. Our IR-based approach computes a numeric score on how much each source code entity is similar to the query (the code example). Then, we rank the candidate set, *e.g.*, all the methods in the system, according to their cosine similarity. The top ranking entities are then considered as *candidate locations*.

Again, consider the case on PackageManager II in which a developer transforms the method platform() in the class GreasePharo30CoreSpec. Table 6.10 shows the top ranked entities in comparison with this method.

6.4.2.4 Replayable Approach

Given a list of candidates for transformation, it is not clear for a code search approach whether the transformations can be actually replayed in each candidate location. To validate their recommendations, we propose to use the macro (when it is available) and try to replay it. Concretely, we extended MacroRecorder to return a binary value indicating whether it was successful in replaying the macro in the code location.

The replaying operation will fail if: (i) an exception is thrown during the transformation, *i.e.*, the code entity to be transformed could not be retrieved in the candidate location; or (ii) the transformations in the macro produced code that was not compilable, consequently MacroRecorder rolls back the all the changes done

Table 6.10: Top-10 most similar methods to GreasePharo30CoreSpec,platform().

Method Signature	Cosine Similarity
SeasidePharo20CoreSpec.platform()	0.912
SeasideTestsPharo20CoreSpec.platform()	0.901
SeasidePharoWelcomeSpec.platform()	0.889
SeasidePharoDevelopmentSpec.platform()	0.839
SeasidePharo20ToolsWebSpec.platform()	0.836
SeasidePharoToolsSpecSpec.platform()	0.836
SeasideCanvasPharo20Spec.platform()	0.835
SeasideCanvasPharo30Spec.platform()	0.835
SeasidePharoEmailSpec.platform()	0.829
JavascriptPharo20CoreSpec.platform()	0.826

by the macro. In such cases, we assume the *candidate location* was an incorrect one and we remove it from the list of recommendations.

It is worth noting that MacroRecorder does not perform the transformations immediately on code. The tool first performs them on a model to check preconditions and display the modified code to the developer, who will ultimately accept or reject the modifications.

6.5 Validating Recommendations

In this section, we evaluate precision and recall of the code search approaches proposed in this thesis. Section 6.5.1 presents how we compute candidate locations for the transformations. Then, we describe the metrics we used in this evaluation in Section 6.5.2. We evaluate structural, AST-based, and IR-based approaches in Section 6.5.3. We evaluate our fourth approach, *i.e.*, using the macro to validate the recommendations, in Section 6.5.4. We discuss two approaches that compute ranked recommendations, namely AST-based and IR-based approaches, in Section 6.5.5. Finally, we present threats to validity in Section 6.5.6.

6.5.1 Finding Candidates for Transformation

Our dataset consists of the macros we recorded in Chapter 5. Our goal is to compute candidate locations for transformation with these macros, using the approaches we proposed in Section 6.4.2. Additionally, as discussed in Section 6.4.2, our approaches require some input which is retrieved from our dataset as follows.

- The source code under analysis are indicated in Table 5.1 (page 63). Only the versions before rearchitecting are considered for analysis. All classes and methods of the systems are used as input, *i.e.*, all code entities are considered potential candidate locations.

- We used the macros as recorded by MacroRecorder in Chapter 5 in Section 5.4.3. The macro is used as our *"fourth"* approach to filter the candidate list by dropping those candidates where the macro cannot be replayed (Section 6.4.2.4). We assume at this point that the macro is configured, *i.e.*, it was already replayed to at least one more location. This way, it is expected that the macro will not reject correct candidate locations from the candidate list.
- Our approaches require one (for AST-based and IR-based) or more (for structural) code examples. These code examples are selected randomly from all the actual occurrences of the macro (see again Table 5.2). To alleviate the threat that the result of code search might depend on the selection of code examples, we execute the approaches several times with different code examples. We report in this thesis results in which the selection produced most candidate locations.

Each approach will generate a list of candidates for transformation from which we can compute precision and recall according to our oracle.

6.5.2 Evaluation Metrics

In this section, we present the metrics we use in the evaluation. Our approaches return a list of candidate locations as a result (the $Candidates$ set). For each instance of macro, the oracle set represents the code locations that were in fact modified by the developers (the $Correct$ set).

Precision is the percentage of identified candidates that are correct. Recall measures the percentage of correct locations identified by a given approach. These metrics are also described more formally as follows:

$$precision = \frac{|Correct \cap Candidates|}{|Candidates|} \tag{6.1}$$

$$recall = \frac{|Correct \cap Candidates|}{|Correct|} \tag{6.2}$$

Typically, a better recall comes with lower precision, and vice-versa. On one hand, recall is important because we want to avoid omissions, *i.e.*, the approach should be able to find all the correct transformation opportunities. On the other hand, as a recommendation tool for the developer, it is also important that the approach returns as little incorrect candidates as possible (*i.e.*, a high precision). Therefore, we aim for higher precision rather than higher recall.

On top of these two metrics, we added a third one for AST-based and IR-based approaches. Both rank their list of candidates in decreasing order of similarity.

In this case, special ranking metrics, such as the Discounted Cumulative Gain (DCG) [JK00], were proposed by practitioners to weight correct recommendations based on their ranking position. Concretely, DCG weights correct results near the top of the ranking higher than in lower positions of the ranking. The assumption is that a developer is less likely to consider elements near the end of the list.

In the following formula, rel_i indicates the relevance of an entity at rank i and decreases as i augments.

$$DCG_p = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{\log_2(i+1)} \tag{6.3}$$

We compare the AST-based and IR-based approaches using this metric. It is worth noting that DCG is not normalized. Therefore, we only compare both approaches under the same setting, *i.e.*, for the same system under analysis. Moreover, the metric is cumulative; it increases as more candidates are provided. Therefore, we can only compare both approaches under the same candidate list as well.

6.5.3 Overall Results

Table 6.11 presents precision and recall values for the approaches we proposed in this chapter.

Table 6.11: Structural, AST-based, and IR-based results. *Occ.*: number of occurrences of the oracle (as shown in Table 5.2); *Prec.*: precision; *Rec.*: recall. Highlighted cells indicate the best precision results for each macro.

	Occ.	Structural		AST-based		IR-based	
Macro		Prec.(%)	Rec.(%)	Prec.(%)	Rec.(%)	Prec.(%)	Rec.(%)
PetitDelphi	21	12	100	4	100	2	4
PetitSQL I	6	24	100	27	100	16	66
PetitSQL II	98	40	100	32	100	94	32
PackageManager I	66	100	100	74	100	92	89
PackageManager II	19	100	100	100	100	54	100
PackageManager III	64	100	100	100	100	87	100
PackageManager IV	7	66	57	66	57	5	100
MooseQuery I	16	41	100	34	100	19	100
MooseQuery II	8	12	20	2	80	16	10
Pilar	99	19	100	77	100	75	100
Average		51	88	51	94	46	70

Concerning precision, we observed that Structural approach performed as well as AST-based approach, with an average precision of 51% for both approaches. Although the Structural approach only considers package, class, and method names, it performed reasonably well in comparison with the other approaches, including three cases in which all recommendations are correct, *i.e.*, 100% precision. IR-based approach achieved an average precision of 46%, and it had the worst results for some cases such as PetitDelphi (2% precision) and PackageManager IV (5%).

We also observed overestimation with the Structural approach as well. In four cases, less than 25% of candidate locations are correct. For example, in PetitDelphi, developers systematically removed methods of one class which represented a specific grammar rule. The structural approach recommended all the methods of this class as candidate locations, whether or not they did represent this grammar rule. This behavior occurs mainly because the structural approach does not look at the AST. Similar situation occurred in PetitSQL II.

In PackageManager IV and Pillar, some candidates were not found because they were contained in other package than the one from the examples. These cases are exceptions, as can be seen by the good recall.

Concerning recall, the structural approach also gives good results with an average recall of 88%, and eight out of ten cases with 100% recall. AST-based and IR-based approaches achieved an average recall of 94% and 70% respectively.

Regardless of the lower precision and recall, both AST-based and IR-based approaches raised important scalability issues. For example, performing code search around one thousand methods in PetitDelphi (a medium system in our dataset) took more than 15 minutes. It turns out that comparing source code ASTs or processing identifiers takes too long to deploy such approaches into the development environment.

Summary. The Structural approach gives good results concerning both precision and recall. It performed as well as AST-based approach, although the latter raised some scalability issues with medium to large systems. These results indicate that repetitive transformations usually affect similar code locations, *e.g.*, classes in the same package, with the same superclass, or methods in the same class.

6.5.4 Replayable approach results

In this section, we evaluate our replayable approach, *i.e.*, we validate the candidate locations computed by structural, AST-based, and IR-based approaches by trying to replay the macro in each location. We report only precision results because it is expected that macros are configured to be replayed on correct candidate locations. Therefore, the replayable approach does not affect recall.

Table 6.12 shows that the precision increased in four out of ten cases, with two very significant increases observed in PetitSQL I (from 24% to 85%), and Pillar (from 19% to 93%). Similarly to PetitDelphi and PetitSQL, the structural approach recommended all the methods in the hierarchy of document classes in Pillar. We manually inspected each case and, although MacroRecorder could replay the macro, the resulting code would have been incorrect.

Table 6.12: Replayable approach results. Precision results without replayable approach were presented in Table 6.11. Highlighted cells indicate the best precision results for each macro.

Macro	Structural +Replayable Prec.(%)	AST-based +Replayable Prec.(%)	IR-based +Replayable Prec.(%)
PetitDelphi	12	4	2
PetitSQL I	85	85	57
PetitSQL II	40	32	94
PackageManager I	100	74	92
PackageManager II	100	100	54
PackageManager III	100	100	87
PackageManager IV	66	66	5
MooseQuery I	66	48	31
MooseQuery II	40	10	33
Pilar	93	93	93
Average	70	60	54

Summary. Although simple, the Structural-Replayable approach gives very good results with an average precision of 70%. The replayable filter is also easy to implement, when there is a record-and-replay tool availabel, and improves precision for all the other approaches.

6.5.5 Combining Structural with AST-based and IR-based approaches

In particular cases, *e.g.*, Pillar, we observed that AST-based and IR-based approaches performed better than the structural one, despite some performance issues. However, the structural approach performed better and required less resources in most of the systems. In this section, we use the list of candidates generated by the structural approach as the candidate set for AST-based and IR-based approaches (instead of the entire system). For this analysis, we focus on the results of structural approach before validation with the macro.

Table 6.13 presents ranking results, using precision of the candidates at the top-20 position and DCG metric. The DCG metric measures the entire ranking. In the majority of the cases, the AST-based approach produced better rankings than the IR-based one. The precision of the top-20 recommendations was higher in five cases, and three with a tie. In the second and third macros of PackageManager, all the candidates calculated by the structural approach are correct. Therefore, it is expected that the ranked candidates will have the same precision.

Table 6.13: AST-based and IR-based precision and DCG results. *P-20*: precision of the top-20 candidates. Higher DCG is best.

	P-20 (%)		**DCG**	
Macro	AST-based	IR-based	AST-based	IR-based
PetitDelphi	10	10	2.19	2.05
PetitSQL I	30	28	3.31	1.52
PetitSQL II	100	85	20.41	19.26
PackageManager I	100	100	15.61	14.78
PackageManager II	100	100	6.81	6.34
PackageManager III	100	100	15.28	13.94
PackageManager IV	25	30	2.89	1.54
MooseQuery I	70	40	5.62	3.72
MooseQuery II	30	10	1.87	0.82
Pilar	95	90	20.46	19.86
Average	66	59	9.38	8.38

Concerning the DCG metric, which measures the entire ranking, AST-based approach performed better than the IR-based one in all of the cases. In most of the systems, a higher precision for either AST or IR-based approach also implied a higher DCG. In PackageManager I, the AST-based approach have lower precision but higher DCG. Since DCG is a cumulative metric, its result indicates that the AST-based approach places correct recommendations (after the 20^{th} position) in a higher position in comparison with the IR-based approach.

However, both approaches did not improve PetitDelphi's precision. In Section 6.5.3, we discussed that developers removed all the methods representing a particular grammar rule. This particularity is represented by using the operator "," (a comma).[3] The IR-based approach does not consider this operator as a term; instead, the similarity considered only the name of the method. Moreover, the AST-based approach only produce high similarity for methods with the same number of comma operators.

Other limitations appeared when the candidates have few properties in common. For example, in MooseQuery II, the methods transformed by the macro have short names (*e.g.*, from and to), and they only share one return statement in common. Thus, both AST and IR similarities will be low even between correct candidates; the recommendations will then be sorted with incorrect ones. Similar cases occurred in PetitSQL I and PackageManager IV.

[3]Pharo allows one to override operators such as "," or "=".

Summary. The AST-based approach produced more correct ranking in comparison with the IR-based approach. Most of the top-20 recommendations ranked by AST similarity were correct.

6.5.6 Threats to Validity

6.5.6.1 Construct Validity

The construct validity is related to whether the evaluation measures up to its claims. In our evaluation, we use the occurrences of the transformation pattern as our oracle. However, we discussed in Chapter 4 in Section 4.4 that developers missed some candidate locations as well. We also acknowledge this threat. In a practical setting, the list of candidates that our approach produces, either selected by the macro or ranked by AST or lexical similarity, is shown to the developer as a recommendation. Our approach still requires the developer to accept (or reject) the recommendation, either it will be a surprising recommendation or not.

6.5.6.2 Internal Validity

The internal validity is related to uncontrolled factors that might impact the experiment results. In our study, the developers of the systems under analysis are members of our research group. One could assume that results are less significant, because we designed a searching process looking at our own source code. While the identification bias is relevant, it does not affect the essence of the study. The repetitive transformations we found occurred before our study, and therefore they were not influenced by our approach. Our participation in the development only helped us to re-discover them.

6.5.6.3 External Validity

The external validity is related to the possibility to generalize our results. Most of the systems under analysis are small. One may argue that it is easier to find candidate locations in a smaller system. Again, we discussed in Chapter 4 in Section 4.4 that developers missed some candidate locations even in a small system such as PetitDelphi. Moreover, the cases in PetitSQL and Pillar, considered as small, seem to indicate that the size is not an issue. The macros we found in these systems repeated 98 and 99 times, respectively.

6.6 Summary

During a systematic code transformation effort, replaying a sequence of code transformation requires the developer to indicate to the code transformation tool, where is the code to be transformed in each occurrence of a transformation pattern. Additionally, it is also required from the developer to select all the code entities that are candidate for automatic transformation. We discussed challenges on performing these tasks in Chapters 4 and 5.

In this chapter, we presented two approaches to automate the process of replaying sequences of code transformations, *i.e.*, macros.

The first approach focused on automating the configuration of the macro when a new code entity is selected to replay the transformations. The approach tries to match the selected entity with the one where the macro was recorded. The parameters of the macro are then configured according to this matching. We reiterate here the most interesting results we found when evaluating this approach.

- Considering all combinations of executions with MacroRecorder, 59% of the parameters were automatically configured by this approach. However, this approach did not provide full configuration in all the macros we recorded, *i.e.*, at least one parameter still require manual configuration from the developer.
- Considering whether a parameter was configured or not, 60% of the parameters were configured correctly by this approach. This result means that, in almost two-thirds of all the occurrences of a transformation pattern, the parameters are configured correctly by our approach. The evaluation leads to the conclusion that customizable, system specific code transformations can be automated with the assistance of developers.

The second approach presented different code search techniques to retrieve similar code entities, from one or more code examples, which are candidate for transformation. This approach requires one or more examples where the macro was applied, *i.e.*, where the macro was recorded or replayed. Additionally, the macro itself is also required to validate the list of candidate locations retrieved by the proposed code search techniques. We reiterate the most interesting results we found evaluating this approach.

- Simply filtering code entities based on their location, *e.g.*, classes in the same package, with the same superclass, or methods in the same class, already produced considerably good results in terms of precision (51%) and recall (88%).
- Additionally filtering the candidate list using the macro, *e.g.*, by testing whether the macro can be replayed in each candidate location, improved precision results in most of the cases. The combination of Structural and Replayable approaches produced better precision results.
- Finally, ranking the candidate list by an analysis of similar AST nodes improved precision results (of the top-20 candidates) in three out of ten cases.

The combination of Structural and AST-based approaches produced better ranking results.

For future work, we propose to allow the developer to edit the transformation operators in cases where they should be performed with small variances. For example, to allow the developer to customize preconditions in the transformation operators. We discussed examples in our case study that MacroRecorder can overcome in order to improve its accuracy. In this case, MacroRecorder would not limited to perform the sequence of transformation operators exactly as they were recorded.

7 Improving Code Transformations

Contents

7.1 Introduction

In the previous chapters, we proposed automated support for repetitive sequences of code transformations that can be applied to the system at large. Due to such systematic impact, it is expected that the system undergoes some intermediate and unstable state until the evolution is completely performed.

In this chapter, we consider code transformations in a more elaborated flow of actions. For example, developers were checking, after moving a class to another package, whether its subclasses should also be moved. If so, additional transformations, *i.e.*, move each subclass, should be performed. These additional transformations are (i) specific for the first performed transformation[1]; (ii) manual, and (iii) repetitive, therefore they would require some automation.

Moreover, we identified that checking whether the hierarchy of a class is placed in the same package is a special case of a design smell, namely Scattered Functionality (as presented in Chapter 3, Section 3.5.1). Design smells may not produce errors, therefore they might remain latent in the system until a major preventive effort takes place (see Chapter 3, Section 3.5.2). In the case when such transformations have to be performed systematically, either manually or using automated support such as MacroRecorder, design smells could also be systematically introduced to the system. Consequently, future correction of these design smells would be compromised: the longer the smell remains latent in the system, the higher the cost to correct it [LGS13].

We propose a tool to recommend additional transformations after a specific transformation is performed, *e.g.*, *Move Class*. This support consists in monitoring code transformations performed by the developer in a development tool. In the specific example of design smells detected when a transformation is performed,

[1]As opposed to system-specific transformations that we discuss in this thesis.

the tool recommends additional transformations to correct it. This is only one example of possible applications of such recommendation approach. Other applications include the correction of bugs that might be introduced, tests that might fail, or compilation errors that might be introduced after a transformation is performed. The transformations as well as the design smells discussed in this chapter were identified by inspecting the flow of actions performed by developers in the systems under analysis.

The main contributions of this chapter are summarized as follows.

- code transformations are self-aware of potential quality violations that they might introduce. In particular, we identified code transformations that might introduce design smells; this association is supported by evaluation in real systems.
- we propose a tool that recommends additional transformations after a code transformation is performed and, particularly, might introduce design smells. We evaluated the recommended transformations with real users when they were performing two particular and well-known refactoring transformations: *Move Class* and *Extract Method*.

Structure of the Chapter

Section 7.2 presents our approach that recommends additional transformations after a design smell is detected. Section 7.3 presents two case studies in which we evaluate the recommended transformations in real cases. Section 7.4 presents threats to validity, and Section 7.5 concludes the chapter.

7.2 Improving Code Transformations

In this section, we introduce our approach to recommend additional code transformation to the developer. Using our approach in practice, the developer performs code transformations in the development IDE, either manually or using automated transformation tools. Each code transformation is stored as a first-class object, from which one can inspect the class/method that was transformed, for example see Section 7.2.1.

After a transformation is performed, the approach tries to detect design violations. In this study, we focused on the detection of design smells. However, other analyses can be performed to check whether the system is in an undesired state. In this thesis, we identified (non-exhaustively) code transformations that are more likely to introduce a design smell. We discuss some examples in Section 7.2.2.

Finally, when a design smell is detected, we suggest a sequence of transformations to remove this smell. We present examples of design smell correction in Section 7.2.3. Considering the number of existing code transformations and design smells, we focus on two particular transformations, *e.g.*, *Move Class* and *Extract Method*. The design smells we study were presented in Chapter 3, Section 3.5.1.

Selection Criteria. The transformations we study in this chapter were systematically performed by developers in our case studies (Section 7.3). We then collected information on these transformations for analysis. Moreover, the rearchitecting in these case studies were *simple* in the sense that only one code transformation was repeatedly performed, *e.g.*, *Move Class* and *Extract Method* separately in different systems. Therefore, the use of our MacroRecorder tool was not required for this study. We discuss issues on the generalization of this approach on Threats to Validity in Section 7.4.

Concerning the design smells we studied, they were motivated by the flow of actions the developers were performing. As discussed in the beginning of this chapter, developers were manually checking a special case of Scattered Functionality design smell. For this reason, we implemented the detection of this smell, and the recommendation of transformations to fix it, specifically for this system. We also discuss this issue on Threats to Validity in Section 7.4.

7.2.1 Recording Code Transformations

Our approach requires a *"recorder"* tool to monitor code transformations performed by the developer. We rely on Epicea [DBG$^+$15], the same tool we use in MacroRecorder to record code transformations from the Pharo IDE, as presented in Chapter 5 in Section 5.3. This monitoring process operates in the background, while the developer is editing the code or performing transformations automatically. Similarly in the Java world, Mylyn [KM06] and COPE [NCDJ14] are other examples of recording tools.

We present the scenarios in which Epicea records the transformations we use in this chapter as follows. Both scenarios are activated by events in the development IDE, therefore tools that infer refactoring transformations from code edition, such as RefFinder [KGLR10], are not required in our approach.

Move Class. This transformation consists in moving a class C from a package P_A to another package P_B. In the IDE, this transformation might be activated either (i) by a code browser menu *"Move to package"* then indicating the target package in a second menu, or (ii) by manually dragging and dropping the class to the target package in the code browser.

Extract Method. This transformation consists in extracting a piece of code into a new method M in the same class. The extracted code is replaced by an invocation to this new method. In the IDE, this transformation is activated via a menu in the code browser, *e.g.*, select the code to be extracted, then select *"Extract Method"*.

Particularly in Epicea, the Extract Method transformation event also stores: (i) an Add Method event corresponding to the extracted method, then (ii) a sequence of Code Replacement events corresponding to the replacement of old code with an invocation to the extracted method.

7.2.2 Checking Design Smells

After a code transformation is recorded in the IDE, our approach performs the detection of design smells. In this case, the approach detects not only the introduction of a design smell, but it also checks whether design smells already existed in the first place.

There are two main considerations in the detection phase. First, the detection takes place locally in the modified code, instead of detecting the smell in the entire source code of the system. For example, after moving class C, we detect design smells in C and optionally in other classes related to this one. And second, only a subset of design smells are checked for each transformation, instead of checking the entire catalog of design smells in each modified location.

In this section, we present our heuristics to detect the design smells we introduced in Chapter 3, Section 3.5.1. We also discuss why the transformations we record in this chapter might introduce these smells.

Scattered Functionality. This smell appears when a single concern is scattered across multiple components. In this study, we consider components as modules, or packages; and we focus on one type of dependency to represent a concern: inheritance. This smell then indicates that a subclass of a class C is placed in a different package from C. The detection of this smell is performed for a given class C as follows.

- C is discarded from analysis if it is a class provided by the system, *e.g.*, a collection or a stream, or provided by a framework. These dependencies are considered acceptable [GPEM09].
- Then, all subclasses of C that are not located in the same package are detected as design smells.

The automatic detection thus identifies classes of the application whose superclasses also belong to the application (not a framework) and are located in a different package.

Rationale. Performing a *Move Class* transformation, *i.e.*, moving a class C from a package P_A to another package P_B might negatively affect the modularization of the system. More specifically, all subclasses of C might be left in the former package P_A. Consequently, the concern represented by the hierarchy of C will be scattered in two packages, P_A and P_B. In practice, after a *Move Class* transformation is performed, we check Scattered Functionality smell on the moved class C.

Unfactored Hierarchy. Considering duplicated code in a type hierarchy, this smell can manifest in two ways: (i) sibling types have similar code that can be pulled up to one of their supertypes; and (ii) super and subtypes have similar code which indicates redundancy, as presented in Chapter 3, Section 3.5.1. We focus on the latter: given a method M in a class C, we perform an Abstract Syntax Tree (AST) search, described as follows.

- The AST representing method M is normalized, obtaining M_n. Nodes representing attributes, arguments, temporary variables, and values are marked as wildcards; return declarations are discarded from the tree.
- Then we search, among all the methods in C and its subclasses, which ones match M_n. A tree matching algorithm proposed by Meng *et al.* [MKM13] is used in this process; we do not focus on the implementation of this algorithm. The matching methods are detected as design smells.

We come back to the example we presented in Chapter 3, Section 3.5.1. Listing 7.1 presents a method iconNamed(String) which code is duplicated in one of the subclasses of Model class.

Listing 7.1: Example of a method iconNamed(String) which code is duplicated in a subclass (highlighted in the second method).

```
class Model {
  public iconNamed(String iconName) {
    return Smalltalk.ui().getIcons().iconNamed(iconName);
  } //... }

class GLMUIThemeExtraIcons extends Model {
  // method with similar code
  public instVarRefactoringMenu(builder) {
    return (builder.newItem("Remove"))
        .setParent("Inst Var Refactoring")
        .setIcon(Smalltalk.ui().getIcons().iconNamed("removeIcon"))
        .setOrder(200);
  }  //... }
```

Considering the method Model.iconNamed(String) was extracted in Model class, we perform the detection of Unfactored Hierarchy on the method iconNamed(String). Therefore, we aim to detect other methods in the hierarchy of Model class that should invoke iconNamed(String). The node representing the argument (iconName) is normalized, and the automatic detection will search all the methods containing (i) a reference to class Smalltalk, followed by (ii) an invocation to the methods ui(), getIcons(), and iconNamed(String) respectively, independent of the argument.

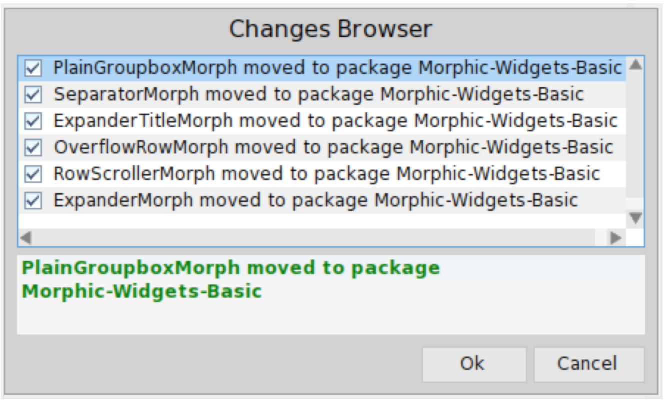
Rationale. After adding a method M, particularly as consequence of an Extract Method transformation, there might be other places in code which also could invoke M. In Eclipse IDE, the Extract Method transformation replaces duplicated code throughout the entire file (*i.e.*, the extracted method class). However, this detection is not done for the subclasses. Concretely, after an *Extract Method* transformation is performed, we check Unfactored Hierarchy smell on the recently created method M.

7.2.3 Recommending Code Transformations

When a design smell is detected, our approach proposes a correction, *i.e.*, a sequence of transformations which will remove the smell. Suryanarayana *et al.* [SSS14] proposes high-level suggestions and a desired design which does not contain a given design smell. However, the authors mostly do not propose the sequence of transformations that shall be performed into smelly design to become the desired one.

In this section, we present the sequence of transformations for fixing design smells checked in Section 7.2.2. These sequences were defined programmatically, making use of the transformation tools available in Pharo.

Scattered Functionality. This smell indicates that a subclass of a class C is placed in a different package from C. The correction of this design smell consists in moving this subclass to the same package of C. The computed transformations are shown to the developer as recommendations. Figure 7.1 shows the sequence of transformations proposed to correct the smell. The subclasses were found by the Scattered Functionality smell detection. The developer can either accept or reject each proposed transformation.

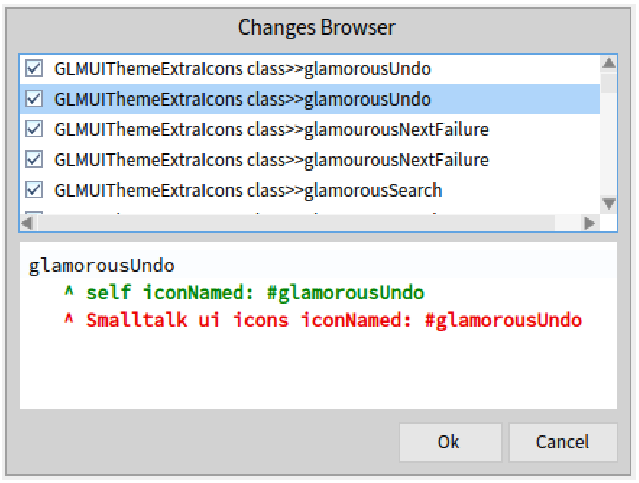


- ✓ PlainGroupboxMorph moved to package Morphic-Widgets-Basic
- ✓ SeparatorMorph moved to package Morphic-Widgets-Basic
- ✓ ExpanderTitleMorph moved to package Morphic-Widgets-Basic
- ✓ OverflowRowMorph moved to package Morphic-Widgets-Basic
- ✓ RowScrollerMorph moved to package Morphic-Widgets-Basic
- ✓ ExpanderMorph moved to package Morphic-Widgets-Basic

PlainGroupboxMorph moved to package Morphic-Widgets-Basic

Figure 7.1: Sequence of transformations proposed to the developer. A preview of the changes is shown in the lower panel, which the developer can accept or discard.

Unfactored Hierarchy. This smell indicates that the source code of a method M is partially implemented in another method in the same class or its superclasses, indicating code duplication. The correction of this design smell consists in replacing the duplicated code in M by an invocation to the method M_i which already implements the duplicated code. The calculated transformations are shown to the developer as recommendations. Figure 7.2 shows the sequence of transformations proposed to correct the smell.



```
☑ GLMUIThemeExtraIcons class>>glamorousUndo
☑ GLMUIThemeExtraIcons class>>glamorousUndo
☑ GLMUIThemeExtraIcons class>>glamourousNextFailure
☑ GLMUIThemeExtraIcons class>>glamourousNextFailure
☑ GLMUIThemeExtraIcons class>>glamorousSearch
```

```
glamorousUndo
    ^ self iconNamed: #glamorousUndo
    ^ Smalltalk ui icons iconNamed: #glamorousUndo
```

Figure 7.2: Sequence of transformations proposed to the developer. A preview of the changes is shown in the lower panel, which the developer can accept or discard.

7.3 Validation Experiment

In this section, we evaluate our approach for recommending additional transformations after a design smell is detected. We measure the precision of our approach to check whether the recommended transformations are accepted by the developer. Moreover, we measure recall to check how many transformations are automated by our approach, *i.e.*, the developer did not need to perform them manually.

We separate this evaluation in two studies. In the first study, we replay a sequence of *Move Class* transformations performed in a real case of module decomposition (Section 7.3.1). The second study is conducted with a developer performing *Extract Method* transformations with our approach, in a real case of adaptive maintenance (Section 7.3.2).

7.3.1 Case Study I – PolyMorph

In this study, we investigate a substantial maintenance effort in a real project. PolyMorph is an extending layer for the default user interface framework of Pharo, named Morph. It contains more than 20 new widgets and it introduces new visual effects, while it systematically added patches to the system. The project was designed and packaged monolithically with the hypothesis that Morph could not be changed to enable extensions such as PolyMorph. PolyMorph was then integrated in Pharo with the goal to revisit the underlying UI framework.

The maintenance in this project consisted in decomposing the biggest package, PolyMorph-Widgets, into smaller ones. The new packages were named Morphic-Widgets-* where the last name denominates a set of similar widgets, e.g., Tabs, Scrolling, Basic, etc. Classes from other packages, which implemented similar UI widgets, were also included into this new organization. Consequently, numerous *Move Class* transformations were performed, either manually or with automated support, in not one but several versions.

Data Collection. We retrieved the version 3.0 of Pharo, released in May 2014. In this version, the package PolyMorph-Widgets was still monolithic. Then, we chose the latest Pharo version (6.0), under development since May 2016, as our target version. Table 7.1 presents information about this project.

Table 7.1: Descriptive data about PolyMorph project. The number of classes decreased because some were moved to packages other than Morphic-Widgets. These transformations are out of the scope of this study.

	Pharo v3.0	Pharo v6.0
Number of Packages	9	20
Number of Classes	963	492
KLOC	177	145

We computed: (i) classes that were moved from package PolyMorph-Widgets to any of Morphic-Widgets-* packages; and (ii) classes already existing in the system that eventually were moved to one of Morphic-Widgets-* packages. In Pharo, the class container is specified in the class declaration. We detected classes that were moved by identifying modification in the class meta-information. Classes created and eventually moved during the evolution of PolyMorph, *i.e.*, between versions 3.0 and 6.0, are discarded from this analysis. A total of 132 classes were moved to one of the new Morphic-Widgets packages from versions 3.0 to 6.0.

Measuring Accuracy. We performed each of the 132 *Move Class* transformations using our approach. In this study, we aim to compute all the recommendations that our approach can propose.

Each transformation might generate recommendations, *i.e.*, suggest to additionally move more classes when a Scattered Functionality design smell is detected. Additional transformations might affect future recommendations and therefore the recommendations are not effectively accepted. A recommendation, *e.g.*, move class C to package P_B, is considered correct if class C was originally moved to package P_B in the code history of PolyMorph, *i.e.*, the transformation is one of the 132 we identified. Hence, we measure:

- **Precision** is the percentage of recommendations that were performed previously in PolyMorph.
- **Recall** measures the percentage of correct recommendations identified over all (132) moved classes in PolyMorph. In other words, recall measures the percentage of the originally moved classes that would be automatically moved using our approach.

Results. Table 7.2 summarizes the results in this case study.

Table 7.2: Recommending additional transformations in PolyMorph. Values between parenthesis represent recommendations that were not applied in PolyMorph's code history, but later validated with developers.

#Transformations	132
#Recommendations	79
#Correct	43 (+19)
Precision	78%
Recall	32%

Our approach recommended 79 Move Class transformations in PolyMorph. More than a half (43 out of 79) of these recommendations were performed during the code history of this project, and therefore they were considered correct. This result also means that 36 classes were detected as Scattered Functionality in the project, and they were left uncorrected.

We consulted with developers on these 36 classes:

- Four classes were completely removed from the system in version 6.0;
- Seven classes were moved to Morphic-Widgets-* packages other than their superclasses' because they represented specific widgets. The developers then created a separated package for these widgets;
- Five classes were suggested by developers to move to other packages because these classes represented basic widgets, *i.e.*, commonly composed into more complex ones. Developers moved these classes to a *core* package;

- One class was marked as bad use of inheritance, *i.e.*, it should be a composition of widgets and not a subclass of one. The recommendation thus indicated this bad use of design;
- Finally, the remaining 19 recommendations were considered correct by the developers, as they acknowledged that the decomposition of PolyMorph was still in progress. Therefore, we count $(43 + 19)$ 62 correct recommendations, and a precision of 78%.

Concerning recall, almost one third (43 out of 132) of *Move Class* transformations could be automated by our approach, *i.e.*, the developer could have avoided to perform *Move Class* 43 times. Although this transformation has automated support in the IDE, it is required for the developer to explicitly select the class to move and the target package each time. In summary, this case study is a small example of how additional quality checks are important to automate transformations that would be performed repetitively otherwise. More importantly, our approach identified transformation opportunities that would be forgotten by developers, *e.g.*, the 19 additional correct recommendations.

Summary. 78% of the recommendations are considered correct. This result includes 19 recommendations that developers forgot to perform. Moreover, 32% of the transformations were computed automatically, *i.e.*, the developer could have avoided to perform *Move Class* 43 times using our approach.

7.3.2 Case Study II – Revamping Icon Management

In this study, we investigate an ongoing maintenance effort in Pharo. To access icons, developers were using a coding idiom based on global registry that was in fact a disguised access to a global variable. This practice became so common that even experienced programmers forgot that they can define a method to factor such global invocation, limiting the impact of future maintenance.

As a result of this common practice, the invocation to this global registry was duplicated over the entire Pharo ecosystem. It was an explicit concern by the developers to reduce this duplication. Listing 7.2 presents an example of such invocation that previously presented in Section 7.2.2. In this example, Smalltalk is a global dictionary which contains classes, global variables, and all registered icons in the system.

Listing 7.2: Example of a method in GLAMOUR which is candidate for maintenance. It contains a direct reference to a global registry (with "Smalltalk").

```
public instVarRefactoringMenu(builder) {
    return (builder.newItem("Remove"))
        .setParent("Inst Var Refactoring")
        .setIcon(Smalltalk.ui().getIcons().iconNamed("removeIcon"))
        .setOrder(200);
}
```

In this example, the maintenance consisted in extracting the invocation "*Smalltalk.ui().icons().iconNamed(String)*" to a new method: iconNamed(String). Concretely, Extract Method transformations were performed in a bottom-up way: (i) first factor all invocations inside a method, as the one presented in Listing 7.2; (ii) then at the class level, *i.e.*, replace invocations in all the methods of one class; (iii) then at the level of a class hierarchy, *i.e.*, pull up method iconNamed(String) and perform (ii) in the superclass.

Consequently, several code replacements were also performed. Listing 7.3 presents the solution of the example in Listing 7.2. The desired result is as few as possible places that directly invoke such global registry. This way, future versions of Pharo can freely decide how to manage and eventually replace icons.

Listing 7.3: Results of transformations performed in the example presented in Listing 7.2. Other invocations in the same class should be replaced with an invocation to the recently extracted method. The method iconNamed(String) will be eventually pulled up in the hierarchy of this class.

```
public iconNamed(String iconName) {
    return Smalltalk.ui().getIcons().
      iconNamed(iconName);
}

public instVarRefactoringMenu(builder) {
    return (builder.newItem("Remove"))
        .setParent("Inst Var Refactoring")
        .setIcon(this.iconNamed("removeIcon"))
        .setOrder(200);
}
```

The maintenance impacted the entire Pharo ecosystem. The modified code was submitted in slices for continuous integration, and the maintenance was reported as "*Clean Up*" issues in Pharo's issue tracking system.[2] We selected these issues to identify the commits we will analyze in this study.

Data Collection. We retrieved an early version 6.0 of Pharo. Specifically, the version 60033 is the last one before developers started performing this maintenance.

[2]https://pharo.fogbugz.com/

Then, we retrieved the slices in the continuous integration repository.[3] Table 7.3 presents descriptive data about this maintenance effort.

Table 7.3: Descriptive data about Pharo icon management maintenance.

Number of Open Issues	11
Number of Commits	14
Date of First Commit	22-05-2016
Date of Last Commit	23-09-2016
Number of Extract Method	19
Number of Replacements	389

We computed: (i) methods iconNamed(String) added to the system, and (ii) methods in which a replacement from "*Smalltalk.ui().getIcons().iconNamed(String)*" to "*this.iconNamed(String)*" was performed. We identified code replacements manually because we had a reasonable number of commits under analysis, and the replacement itself concerns a couple of lines of code in each method. Up to September 2016, 19 methods were created and 389 code replacements were performed related to this issue.

Measuring Accuracy. We performed each of the 19 *Extract Method* transformations in an early version of Pharo, namely version 60033. These transformations are the ones that recommend additional code replacement transformations. Each transformation might generate recommendations, i.e., suggest to invoke the extracted method in the class and its subclasses. A recommendation, *e.g.*, code replacement, is considered correct if it was performed by the developers and submitted to the code repository, *i.e.*, it is one of the 389 replacements we identified (see Table table:iconnamed).

Results. Table 7.4 summarizes the results in this case study.

Table 7.4: Recommending additional transformations with Extract Method transformation. Values between parenthesis represent recommendations that were not applied in Pharo, but they were later validated with developers.

#Transformations	19
#Recommendations	352
#Correct	278 (+58)
Precision	95%
Recall	71%

Our approach recommended 352 code replacements, *i.e.*, replace "*Smalltalk.-ui().getIcons().iconNamed(String)*" to "*this.iconNamed(String)*" when the class or

[3] `http://smalltalkhub.com/#!/~Pharo/Pharo60Inbox`

superclasses already implement a method iconNamed(String). Almost 80% of these transformations (278 out of 352) were performed during the code history of Pharo and, therefore, they were considered correct. It is worth noting that the replacements are counted per method. For example, the method FileList.contentMenu() contains eleven occurrences of the invocation to "*Smalltalk.ui().getIcons()*". In our analysis, they are counted as one replacement. Our approach replaces all of these occurrences automatically.

Moreover, 74 out of 352 recommendations were not performed by the developer, thus being detected as Unfactored Hierarchy design smell and left uncorrected in Pharo. We consulted with developers on these 74 recommendations:

- Six methods were removed from the system in a latest development version;
- In eight methods, their classes represented a composition of widgets, *e.g.*, a tabbed window is composed of tabs. Developers preferred to delegate the icon instantiation to the containing widget (*e.g.*, a tab) instead of "*this*";
- Two methods iconNamed(String) were detected. This fact means that, after adding a method iconNamed(String) in the superclass, as performed by the developers, our approach identified two identical and existing methods in one of its subclasses. These methods then shall be removed from the system;
- Finally, the remaining 58 recommendations were considered correct by the developers, as they acknowledged that the code replacement was still in progress. Therefore, we count (278 + 58) 336 correct recommendations, and a precision of 95%.

Concerning recall, 71% of the code replacements (278 out of 389) could be automated by our approach, *i.e.*, the developer could have performed 278 code replacements with only 19 Extract Method transformations. As discussed in Section 7.2.2, Extract Method has automated support in the IDE, however the code replacements are limited to the method the developer is transforming (for Pharo), or the file that the developer is editing (for Eclipse).

Summary. 95% of the recommendations are considered correct. This result includes 58 recommendations that developers forgot to perform. Moreover, 71% of the code replacement transformations were computed automatically, *i.e.*, the developer could have avoided to perform this transformation 278 times using our approach.

7.4 Threats to Validity

7.4.1 Construct Validity

The construct validity is related to how well the evaluation measures up to its claims. The design smell detection we implemented in this study are specific cases of other smells defined in other catalogues, such as the one proposed by Fowler *et al.* [FBB+99]. Scattered Functionality is an specific version of Misplaced Class: the former only considers that classes in the same hierarchy and in different packages are misplaced. Moreover, Unfactored Hierarchy is a specific manifestation of Duplicated Code: the former is only detected when the duplicated code is placed up in the hierarchy of the class. One may argue that this implementation is insufficient and/or it is an over specification of well-known smells, which might have more stable techniques to detect them.

We acknowledge this issue. We do not argue that ours is the only implementation for detecting Scattered Functionality, or Unfactored Hierarchy. In fact, our results for PolyMorph showed some false-positives, because some classes were either too specific (move to a new sub-package) or too generic (move to a core package). Other factors, in addition to inheritance, might have been put into analysis in these cases, and other related smell detection tools could have been used. As discussed in Chapter 3, Section 3.5.1, it is expected that not all entities detected as design smells are in fact validated by the developer. Even with a specific implementation, we reported considerably good results. We further discuss this issue in internal validity.

7.4.2 Internal Validity

The internal validity is related to uncontrolled factors that might impact the experimental results. In our evaluation, the transformations performed by the developers were computed from the code history of the systems under analysis. One may argue that our oracle might not accurately represent the transformations that were performed. Both investigated transformations, *e.g.*, *Move Class* and *Extract Method*, have results that are easily detectable using a diff: a class which meta-information was modified and a method named iconNamed(String) that was added, respectively. Only the code replacement was identified manually. However, this threat is alleviated because the replacement was also simple to identify, *i.e.*, it only modified a couple of lines of code in a very similar way (see Listings 7.2 and 7.3).

In our case studies, the developers of the systems under analysis are members of our research group. Indeed, the fact that some of us had knowledge of the transformations involved was a big help. However, it must be noted that some of the transformations occurred before our study, *e.g.*, the ones in PolyMorph which started back in 2014, and the early patches concerning icon management back in

May 2015. Our participation in the development helped us to re-discover both maintenance efforts, and to also provide automated support for the ones that were still in progress.

In our evaluation, we identified classes and methods detected as design smells that were not corrected by the developers. Most of these cases were in fact forgotten by the developers and therefore considered as correct suggestions for transformation. However, some of the smells still remain in the system. In both case studies (PolyMorph and icon management), these cases were justified by the developers and considered as exceptions. For example, creating a new sub-package for part of the class hierarchy, or delegating the icon instantiation to a containing widget, were part of the solution for those systems.

7.4.3 External Validity

The external validity is related to the possibility to generalize our results. In this study, we implemented the detection of two design smells. The detection we implemented was motivated by the maintenance effort under analysis, and therefore it might not generalize to other systems. One may argue that other design smells could be checked after a *Move Class* is performed, for example. Recent studies showed that code anomalies such as design smells often *"flock together"*, *i.e.*, one design smell represents only a part of a bigger design problem [OGdSS+16]. Additionally, we do not consider potential side effects caused by correcting a design smell. For example, moving classes to a new package might introduce undesired dependencies between former and new package, and consequently introduce other smells, *e.g.*, Feature Envy.

We acknowledge both issues. It is not the scope of this thesis to evaluate exhaustively all design smells and/or all the transformations that might introduce them. However, we do not exclude the opportunity for detecting more design smells. In the PolyMorph case, developers were constantly checking whether the widgets hierarchies should be moved altogether to new packages. The design smell check and the following recommendations for transformations thus assisted them to do this job. Future maintenance in Pharo will motivate us to implement additional design smells and their respective correction.

7.5 Conclusions

In this chapter, we presented an approach to recommend additional transformations when developers are transforming code. We compute the recommendations based on the detection of quality violations. This process is done locally, *i.e.*, our approach searches for smells only in the transformed code. Moreover, this process is transparent for the developer, *i.e.*, in the case where our approach detects design smells, it will show a warning with the additional transformations to be performed to correct the smell. Otherwise, the developer may normally resume the transformations.

To validate our approach, we evaluated two code transformations, *e.g.*, *Move Class* and *Extract Method*. Our case studies consisted in real cases of large maintenance effort in which these transformations were heavily performed. We reiterate the most interesting conclusions from our experiment results:

- The majority of the recommendations computed by our approach were considered correct (78% for *Move Class*, and 95% for *Extract Method*). Most importantly, these results include recommendations that were missed by the developers, since both maintenance efforts are still in progress. These missing opportunities might have remained unnoticed and then lead to inconsistent code.
- Our approach was able to recommend transformations that would be otherwise performed "*manually*" by the developer (31% for *Move Class*, and 71% for *Extract Method*). This result reveals that these transformations were part of the developers' maintenance flow, and our approach is able to automate them.

As future work, we plan to extend this research to analyze additional design smells and their automatic correction. This extension will enable us to evaluate more maintenance efforts and to better identify the impact between code transformations and the introduction of design smells.

8 Conclusion

Contents

8.1 As a Conclusion

Software evolution is a complex task. Sometimes, a large rearchitecting effort must be performed; for example, to migrate the system to a new architecture. The impact of rearchitecting may be large and affect the entire system. We need to ensure that changes are consistently applied in the system.

We investigated rearchitecting cases from real-world systems. We found several sequences of code transformations that were performed in a systematic way, *i.e.*, several code entities were transformed in a similar manner. The sequences we found are language independent, *e.g.*, we studied Java and Pharo systems, however they are specific to the systems in which we found them.

Due to the repetitive nature of these transformations, these sequences were tedious to perform. Moreover, manually performing these sequences of transformations is an error-prone task. Developers missed code entities that should have been transformed, or they did not perform all the transformations defined in the sequence. One would benefit from automated support to perform these systematic sequences of transformations.

In this thesis, we argued for the need for automated support of sequences of code transformations. We covered four aspects:

1. the definition and identification of these sequences. We provided examples of system-specific sequences that were manually performed and properties that motivate their automation [SAE$^+$15c].
2. the automatic configuration of these sequences to be further reapplied in other code locations. We provided a proof-of-concept tool to record and automatically replay sequences of code transformations specified by the developer [SAE$^+$15b].
3. the recommendation of code locations that might be candidates for systematic transformation. We provided an approach to recommend code locations on which a sequence of transformations can be performed [SPA$^+$17].
4. the recommendation of additional transformations when the system is put in an undesirable state. We provided an approach to recommend additional

transformations, *i.e.*, a flow of actions to be performed after a code transformation might introduce quality violations (paper in submission for an international conference).

The evaluation included the automatic reapplication of the sequences we discovered, in which research questions were answered, when necessary with the use of statistical tests. The experts also played an important role in our study, by suggesting examples of systematic transformations that were done in the past, and by evaluating the recommendation of additional transformations in rearchitecting efforts that were still in progress.

We now present a summary and we reiterate the most interesting conclusions we derived from our study.

Relevance of Systematic Code Transformation

This work reported on an investigative study we conducted on rearchitecting cases from real systems. The study was performed on seven systems, small to large, and developed on Java and Pharo. We found nine sequences of code transformations in five out of these seven systems; these sequences were systematically performed up to 72 times. To the best of our knowledge, these sequences were manually performed by the developers. These sequences were not always applied to all the code entities that should be transformed. In some cases, developers did not perform all the transformations in the sequence, or all the transformations were not performed in one shot but over several revisions. This fact might indicate that the sequences we found are complex and/or tedious to perform. We reported a description of the sequences we found, *e.g.*, the transformations performed, and the entities modified by them, to propose some automated support [SAE+15c].

Supporting Systematic Code Transformation

In this work, we proposed automated support to avoid errors of omission due to the manual and repetitive application of sequences of transformations, called *macros*. We presented a proof-of-concept tool, in which the developer records the sequence of transformations once, then generalizes the recorded macro to apply it automatically in other code locations. We evaluated our tool using the sequences of transformations we found; and additional sequences were suggested by the Pharo community. Manual configuration was required in three out of five parameters per macro. Our tool was able to perform 88% of the occurrences of the macro with 84% accuracy. The source code resulting from automatic transformation is 94% similar to manual code edition from the developer [SAE+15b].

Automating Systematic Code Transformation

In this work, we proposed an automated support to replay a macro in two ways. The first approach focused on the configuration of a macro. After the developer selects a code location in which he/she wants to replay a macro, our approach tries to match the selected entity with the one where the macro was recorded. The parameters of the macro are then configured according to this matching. We computed all the combinations for recording and replaying a macro, *i.e.*, considering different code examples as input. In 60% of all the executions, the parameters of the macro were configured correctly by this approach. The second approach focused on recommending code locations in which a macro can be performed. We investigated different code search techniques that retrieve similar code entities, given one or more code examples. A basic search on code entities based on their location, *e.g.*, classes in the same package, with the same superclass, or methods in the same class, produced relevant results in terms of precision (43%) and recall (85%). Additionally filtering the candidate list using the macro, *e.g.*, by testing whether the macro can be replayed in each candidate location, improved precision up to 70%. Finally, ranking the candidate list by an analysis of similar AST nodes improved precision results (of the top-20 candidates) in three out of ten cases [SPA+17].

Improving Code Transformations

This work proposed an approach to recommend additional transformations after developers perform transformations that might put the system in an undesirable state. For this study, we computed the recommendations based on the detection of quality violations, *i.e.*, design smells. The study was evaluated with two refactoring transformations, *e.g.*, *Move Class* and *Extract Method*. Our case studies consisted in real cases of large maintenance effort in which these transformations were heavily performed. The majority of the recommendations computed by our approach were considered correct (78% for *Move Class*, and 95% for *Extract Method*). Most importantly, these results include recommendations that were missed by the developers. Our approach was able to recommend transformations that would be otherwise performed manually by the developer (31% for *Move Class*, and 71% for *Extract Method*). This result reveals that these transformations were part of the developers' maintenance flow, and our approach is able to automate them.

8.2 Future Work

There are some open issues that were not addressed in this thesis, and some opportunities for research that should be explored in future work.

Evaluation with Real Developers

In the evaluations with MacroRecorder (in Chapters 5 and 6), we focused on replaying transformation patterns that occurred in the past. Our tool was able to perform these patterns correctly in most of the cases. The fact that most of the macros were modified automatically is a good result. However, we did not rely on real developers effectively using the tool. Based on that, we propose as a future work to monitor real developers using MacroRecorder, which is available for the Pharo community. The goal consists in evaluate the usability of the tool, and also gather other examples of macros.

Allowing the Developer to Customize the Transformations

In this thesis, we identified some cases in which transformation patterns were performed with small variations. Our tool was limited to replay the transformations in the same sequence as they were recorded. We suggest two features for future work: (i) allow the developer to select a subset of transformations that can be internally repeated given a condition, *e.g.*, *while* it exists an invocation to method addPlatformRequirement(String), *do* remove this invocation and substitute it to a literal value; and (ii) allow the developer to disable a subset of transformations when a condition is set, *e.g.*, *if* there is no invocation to method addPlatformRequirement(String), *do* not try to remove invocations. To implement this feature, we propose a DSL to describe the transformations, its parameters, and application conditions; this DSL would be an extension of the one we proposed in Chapter 5.

Automated Support for Other Programming Languages

In Chapter 3, we discussed that existing automated code transformation approaches lack the definition of transformations that are system-specific, eventually complex, and not localized. We focused the implementation of our tool for Pharo systems. However, we also identified transformation patterns on Java systems in Chapter 4, which we did not provide support to replay them. This restriction was done due to the potential adoption of this tool by the Pharo community, which includes our research group, and by the fact that most of the patterns were found on Pharo systems. Still, for generalization purposes, we intend to implement the approaches we presented in this thesis for Java systems. Hence, we would investigate whether the limitations we identified are also present in a statically typed programming language with a more complex grammar.

Improving More Code Transformations

In our study to recommend additional transformations (in Chapter 7), we investigated two refactoring transformations, *e.g.*, *Move Class* and *Extract Method*. Moreover, the recommendations were based on the introduction of design smells. This restriction was done due to flow of actions that developers were performing, which originally motivated the implementation of such recommendations. For generalization purposes, we intend to investigate other code transformations that are most frequently performed in the Pharo community. We also intend to recommend transformations based on other activities, such as fixing bugs that might be introduced, or correcting tests that might fail after a code transformation is performed by the developer.

8.3 Collaborations

During the Ph.D., I had the opportunity to collaborate with the ASERG/UFMG group (Belo Horizonte, Brazil) and the LASCAM/UFU group (Uberlândia, Brazil).

ASERG/UFMG group

In this collaboration, I visited the group once (March/2016). As a result of this visit, we worked with two M.Sc. students on survey studies with real developers, which concerned: (i) their perceptions about AngularJS [RVTS16], and (ii) their perceptions about software architecture documentation and verification [MSSV16]. The group's director, Prof. Marco Túlio Valente, with which we have collaboration since the beginning of this thesis, visited our research group in December/2016. We have mainly worked in our record-and-replay tool, MacroRecorder, which covers most of this thesis. From this collaboration, three papers were published in international conferences [SAE+15a, SAE+15b, SAE+15c] and one paper was submitted (under review) for a journal.

LASCAM/UFU group

In this collaboration, I visited the group once (February/2015). Our research group welcomed a Ph.D. candidate, Klérisson Paixão, for a 10-months internship starting on September/2015. We have mainly worked on our approach to recommend code locations after a macro is recorded. Future work was proposed to implement automated support for Java systems. From this collaboration, one paper was published for an international conference [SPA+17].

Bibliography

[ABFP86] G. Arango, Ira Baxter, Peter Freeman, and Christopher Pidgeon. TMM: Software maintenance by transformation. *IEEE Software*, 3(3):27–39, 1986. 3, 19

[ADW+15] Francesca Arcelli Fontana, Jens Dietrich, Bartosz Walter, Aiko Yamashita, and Marco Zanoni. Preliminary catalogue of anti-pattern and code smell false positives. Research report, 2015. 25

[AGF11] Roberta Arcoverde, Alessandro Garcia, and Eduardo Figueiredo. Understanding the longevity of code smells: Preliminary results of an explanatory survey. In *4th Workshop on Refactoring Tools*, pages 33–36, 2011. 2

[AL08] Jesper Andersen and Julia L. Lawall. Generic patch inference. In *Automated Software Engineering International Conference*, pages 337 – 346, sep 2008. 2, 3, 18

[AL11] Nicolas Anquetil and Jannik Laval. Legacy software restructuring: Analyzing a concrete case. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR'11)*, pages 279–286, Oldenburg, Germany, 2011. 26

[Are04] Gabriela Beatriz Arevalo. *High-Level Views in Object-Oriented Systems using Formal Concept Analysis*. PhD thesis, University of Bern, 2004. 26

[ASH13] Paris Avgeriou, Michael Stal, and Rich Hilliard. Architecture sustainability. *IEEE Software*, 30(6):40–44, 2013. 1

[BAD12] Muhammad U. Bhatti, Nicolas Anquetil, and Stéphane Ducasse. An environment for dedicated software analysis tools. *ERCIM News*, 88:12–13, January 2012. 9

[BBM10] Sérgio Bryton, Fernando Brito e Abreu, and Miguel Monteiro. Reducing subjectivity in code smells detection: Experimenting with the long method. In *7th International Conference on the Quality of Information and Communications Technology*, pages 337–342, 2010. 25

[BDLMO14] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. *Recommendation Systems in Software Engineering*, chapter Recommending Refactoring Operations in Large Software Systems, pages 387–419. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. 23

[Bie06] Matthias Biehl. APL – a language for automated anti-pattern analysis of OO-software. Technical report, University of Waterloo, Canada, 4 2006. 26

[BMMW14] Alessandro Baroni, Henry Muccini, Ivano Malavolta, and Eoin Woods. Architecture description leveraging model driven engineering and semantic wikis. In *11th Conference on Software Architecture*, pages 251–254, 2014. 9

[Boo04] Grady Booch. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison Wesley, 2004. 25

[BPM04] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS: Program transformations for practical scalable software evolution. *26th International Conference on Software Engineering*, pages 625–634, 2004. 4, 20

[BR98] John Brant and Don Roberts. The refactoring browser. In *12th European Conference on Object-Oriented Technology (Workshops, Demos, and Posters)*, pages 549–549. Springer Berlin Heidelberg, 1998. 20

[BR11] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval - the concepts and technology behind search, Second edition*. Pearson Education Ltd., Harlow, England, 2011. 96

[CDMS02] James R. Cordy, Thomas R. Dean, Andrew J. Malton, and Kevin A. Schneider. Source transformation in software engineering using the TXL transformation system. *Information and Software Technology*, 44(13):827–837, 2002. 3, 4, 20

[Cle96] Paul C. Clements. A survey of architecture description languages. In *8th International Workshop on Software Specification and Design*, pages 16–, 1996. 8

[CM14] Alexander Chatzigeorgiou and Anastasios Manakos. Investigating the evolution of code smells in object-oriented systems. *Innovations in Systems and Software Engineering*, 10(1):3–18, 2014. 3, 29

[DAB$^+$11] Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. MSE and FAMIX 3.0: an interexchange format and source code model family. Technical report, RMod – INRIA Lille-Nord Europe, 2011. 9, 63

[DBG$^+$15] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. Untangling fine-grained code changes. In *SANER'15: Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering*, pages 341–350, Montreal, Canada, 2015. (candidate for IEEE Research Best Paper Award). 58, 60, 109

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002. 7

[DP09] Stéphane Ducasse and Damien Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, July 2009. 7

[DR08] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 481–490, New York, NY, USA, 2008. ACM. 18

[DSA$^+$04] Ignatios Deligiannis, Ioannis Stamelos, Lefteris Angelis, Manos Roumeliotis, and Martin Shepperd. A controlled experiment investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 72(2):129–143, 2004. 3, 28

[EKN91] A. Engberts, W. Kozaczynski, and J. Ning. Concept recognition-based program transformation. In *7th Conference on Software Maintenance*, pages 73–82, 1991. 19

[Erl00] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000. 1

[FBB$^+$99] 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. 20, 23, 24, 25, 26, 28, 120

[FGG08] Beat Fluri, Emanuel Giger, and Harald Gall. Discovering patterns of change types. In *23rd International Conference on Automated Software Engineering*, pages 463–466, 2008. 18

[FGSK03] Robert France, Sudipto Ghosh, Eunjee Song, and Dae-Kyoo Kim. A metamodeling approach to pattern-based model refactoring. *IEEE Software*, 20(5):52–58, 2003. 3

[FWPG07] 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*, 33(11):725–743, 2007. 19

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, November 1995. 7

[GPEM09] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Identifying architectural bad smells. In *13th European Conference on Software Maintenance and Reengineering*, pages 255–258, 2009. 24, 25, 26, 110

[HA11] Uwe van Heesch and Paris Avgeriou. Mature architecting - a survey about the reasoning process of professional architects. In *9th Working Conference on Software Architecture*, pages 260–269, 2011. 8

[HD05] Johannes Henkel and Amer Diwan. Catchup!: Capturing and replaying refactorings to support api evolution. In *27th International Conference on Software Engineering*, pages 274–283, 2005. 20, 22

[HDLL11] Lile Hattori, Marco D'Ambros, Michele Lanza, and Mircea Lungu. Software evolution comprehension: Replay to the rescue. In *19th International Conference on Program Comprehension*, pages 161–170, 2011. 22

[HEA$^+$14] 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. 23

[HKI08] Yoshiki Higo, Shinji Kusumoto, and Katsuro Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(6):435–461, 2008. 23

[HL10] L. Hattori and M. Lanza. Syde: a tool for collaborative software development. In *ICSE Tool demo*, pages 235–238. ACM, 2010. 58

[HR98] Rich Hilliard and Tim Rice. Expressiveness in architecture description languages. In *3rd International Workshop on Software Architecture*, pages 65–68, 1998. 8

[HSK06] Shinpei Hayashi, Motoshi Saeki, and Masahito Kurihara. Supporting refactoring activities using histories of program modification. *IEICE – Transactions on Information and Systems*, E89-D(4):1403–1412, 2006. 29, 30

[inf12] InFusion Hydrogen, design flaw detection tool. `https://marketplace.eclipse.org/content/infusion-hydrogen`, 2012. Online; accessed 26 July 2016. 26

[JAP12] Muhammad Javed, Yalemisew Abgaz, and Claus Pahl. Composite ontology change operators and their customizable evolution strategies. In *Workshop on Knowledge Evolution and Ontology Dynamics, collocated at 11th International Semantic Web Conference*, pages 1–12, 2012. 19, 35

[JJ09] Padmaja Joshi and Rushikesh K. Joshi. Concept analysis for class cohesion. In *13th European Conference on Software Maintenance and Reengineering*, pages 237–240, 2009. 23

[JK00] Kalervo Järvelin and Jaana Kekäläinen. IR evaluation methods for retrieving highly relevant documents. In *23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 41–48, 2000. 99

[JMSG07] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering*, pages 96–105, 2007. 27

[JPW$^+$15] Qingtao Jiang, Xin Peng, Hai Wang, Zhenchang Xing, and Wenyun Zhao. Summarizing evolutionary trajectory by grouping and aggregating relevant code changes. In *22nd International Conference on Software Analysis, Evolution, and Reengineering*, pages 1–10, 2015. 2, 18

[KBD15] Jongwook Kim, Don Batory, and Danny Dig. Scripting parametric refactorings in java to retrofit design patterns. In *31st International Conference on Software Maintenance and Evolution*, pages 211–220, 2015. 20

[KBN07] Miryung Kim, Jonathan Beall, and David Notkin. Discovering and representing logical structure in code change. Technical report, University of Washington, 2007. 18

[KG08] Cory J. Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful: Patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008. 28

[KGLR10] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. Ref-Finder: a refactoring reconstruction tool based on logic query templates. In *18th International Symposium on Foundations of Software Engineering*, pages 371–372, 2010. 109

[KM06] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11, New York, NY, USA, 2006. ACM Press. 58, 109

[KN09] Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *31st International Conference on Software Engineering*, pages 309–319, 2009. 3

[KNE92] Wojtek Kozaczynski, Jim Ning, and Andre Engberts. Program concept recognition and transformation. *IEEE Transactions on Software Engineering*, 18(12):1065–1075, 1992. 4

[KNGWJ13] Miryung Kim, David Notkin, Dan Grossman, and Gary Wilson Jr. Identifying and summarizing systematic code changes via rule inference. *IEEE Transactions on Software Engineering*, 39(1):45–62, 2013. 18

[KVGS09] F. Khomh, S. Vaucher, Y. G. Guéhéneuc, and H. Sahraoui. A bayesian approach for the detection of code and design smells. In *9th International Conference on Quality Software*, pages 305–314, 2009. 23

[KZJZ07] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting faults from cached history. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society. 18

[LDDF11] Jannik Laval, Simon Denier, Stéphane Ducasse, and Jean-Rémy Falleri. Supporting simultaneous versions for software evolution assessment. *Journal of Science of Computer Programming (SCP)*, 76(12):1177–1193, May 2011. 10

[Leh80] Manny Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, September 1980. 7

[Leh96] Manny Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, Berlin, 1996. Springer. 1, 2, 4, 7

[LGS13] Hui Liu, Xue Guo, and Weizhong Shao. Monitor-based instant software refactoring. *IEEE Transactions on Software Engineering*, 39(8):1112–1126, 2013. 30, 107

[Lie01] Henry Lieberman. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. 21

[LLH$^+$10] J. Lawall, B. Laurie, R.R. Hansen, N. Palix, and G. Muller. Finding error handling bugs in openssl using coccinelle. In *8th European Dependable Computing Conference*, pages 191–196, 2010. 4, 20

[LR13] Kevin Lano and Shekoufeh Kolahdouz Rahimi. Optimising model-transformations using design patterns. In *1st International Conference on Model-Driven Engineering and Software Development*, pages 77–82, 2013. 3, 20

[LS07] Wei Li and Raed Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, 80(7):1120–1128, 2007. 3, 28

[LT12] Huiqing Li and Simon Thompson. A domain-specific language for scripting refactorings in erlang. In *15th International Conference on Fundamental Approaches to Software Engineering*, pages 501–515, 2012. 20

[LZ05a] Zhenmin Li and Yuanyuan Zhou. PR-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *10th European Software Engineering Conference Held Jointly with 13th International Symposium on Foundations of Software Engineering*, pages 306–315, 2005. 23

[LZ05b] Benjamin Livshits and Thomas Zimmermann. DynaMine: finding common error patterns by mining software revision histories. *SIGSOFT Software Engineering Notes*, 30(5):296–305, September 2005. 18

[Mar02] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, Department of Computer Science, Politehnica University of Timişoara, 2002. 26

[Mar04] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 350–359, Los Alamitos CA, 2004. IEEE Computer Society Press. 23

[MB08] Slavisa Markovic and Thomas Baar. Refactoring OCL annotated UML class diagrams. *Software and System Modeling*, 7(1):25–47, 2008. 3, 20

[MGDM10] N. Moha, Y. G. Gueheneuc, L. Duchien, and A. F. Le Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010. 25

[MHPB09] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *31st International Conference on Software Engineering*, pages 287–297, 2009. 21

[MKM11] Na Meng, Miryung Kim, and Kathryn S. McKinley. Systematic editing: Generating program transformations from an example. In *32nd Conference on Programming Language Design and Implementation*, pages 329–342, 2011. 3, 22, 62, 66, 67, 69

[MKM13] Na Meng, Miryung Kim, and Kathryn S. McKinley. LASE: Locating and applying systematic edits by learning from examples. In *35th International Conference on Software Engineering*, pages 502–511, 2013. 1, 2, 3, 22, 24, 79, 94, 111

[MKOH12] Katsuhisa Maruyama, Eijiro Kitsu, Takayuki Omori, and Shinpei Hayashi. Slicing and replaying code change history. In *27th Interna-*

tional Conference on Automated Software Engineering, pages 246–249, 2012. 22

[MLL05] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: A program query language. In *20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 365–383, 2005. 59

[MLM+13] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang. What industry needs from architectural languages: A survey. *IEEE Transactions on Software Engineering*, 39(6):869–891, 2013. 8, 9, 11

[MNK+02] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin ichi Sato, and Ken ichi. Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proc. of the 8th IEEE Symposium on Software Metrics (METRICS2002)*, pages 87–94, Ottawa, Canada, June 2002. 28

[Moo10] Moose. Importing and Exporting MSE Files. `http://www.themoosebook.org/book/externals/import-export/mse`, 2010. Accessed: 2016-10-18. 9

[MR14] Tim Molderez and Coen De Roover. Automated generalization and refinement of code templates with ekeko/x. In *23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering, Tool Demonstration Track*, pages 53–58, 2014. 22

[MSSV16] Izabela Melo, Gustavo Santos, Dalton Serey, and Marco Tulio Valente. Percepções de 395 Desenvolvedores sobre Documentação e Verificação de Arquiteturas de Software. In *X Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software*, pages 1–10, 2016. 127

[Mye86] Eugene W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986. 94

[NCDJ14] 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. 18, 58, 109

[NCV+13] 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. 2, 21

[Nei84] James M. Neighbors. The draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, 10(5):564–574, 1984. 3

[NNN$^+$13] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, T.N. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. In *28th International Conference on Automated Software Engineering*, pages 180–190, 2013. 1, 18, 35

[NNP$^+$10] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. Recurring bug fixes in object-oriented programs. In *32nd International Conference on Software Engineering*, pages 315–324, 2010. 3, 20, 23, 93

[NNW$^+$10] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to api usage adaptation. In *2010 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 302–321, 2010. 23

[OCBZ09] Steffen Olbrich, Daniela S. Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *3rd International Symposium on Empirical Software Engineering and Measurement*, pages 390–400, 2009. 2, 4, 28

[OGdSS$^+$16] Willian Oizumi, Alessandro Garcia, Leonardo da Silva Sousa, Bruno Cafeo, and Yixue Zhao. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In *38th International Conference on Software Engineering*, pages 440–451, 2016. 121

[OM08] Takayuki Omori and Katsuhisa Maruyama. A change-aware development environment by recording editing operations of source code. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 31–34, New York, NY, USA, 2008. ACM. 22

[PÍ11] Javier Pérez. *Refactoring Planning for Design Smell Correction in Object-Oriented Software*. PhD thesis, University of Valladolid, 2011. 24

[Pal15] F. Palomba. Textual analysis for code smell detection. In *37th IEEE International Conference on Software Engineering*, pages 769–771, 2015. 26

[Pér13] Javier Pérez. Refactoring planning for design smell correction: Summary, opportunities and lessons learned. In *29th International Conference on Software Maintenance*, pages 572–577, 2013. 2, 24

[PKJ09] Kai Pan, Sunghun Kim, and E. James Whitehead Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009. 18

[PTV$^+$10] L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. das Chagas Mendonca. Static architecture-conformance checking: An illustrative overview. *IEEE Software*, 27(5):82–89, 2010. 7

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992. 11

[PZ12] Ralph Peters and Andy Zaidman. Evaluating the lifespan of code smells using software repository mining. In *16th European Conference on Software Maintenance and Reengineering*, pages 411–416, 2012. 28

[RBJ97] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997. 20

[RBJO96] Don Roberts, John Brant, Ralph E. Johnson, and Bill Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, April 1996. 1

[RDGN10] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, Malaga, Spain, June 2010. 38

[RI14] Coen De Roover and Katsuro Inoue. The ekeko/x program transformation tool. In *14th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 53–58, 2014. 20, 22

[RK12] Baishakhi Ray and Miryung Kim. A case study of cross-system porting in forked projects. In *20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012. 1

[RL08] Romain Robbes and Michele Lanza. Example-based program transformation. In *11th International Conference on Model Driven Engineering Languages and Systems*, pages 174–188, 2008. 3, 21, 23, 35

[Rob99] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999. 1

[RS14] Coen De Roover and Reinout Stevens. Building development tools interactively using the EKEKO meta-programming library. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE*, pages 429–433, 2014. 59

[RVTS16] Miguel Ramos, Marco Tulio Valente, Ricardo Terra, and Gustavo Santos. AngularJS in the wild: A survey with 460 developers. In *7th Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 9–16, 2016. 127

[SAE+15a] Gustavo Santos, Nicolas Anquetil, Anne Etien, Stephane Ducasse, and Marco Túlio Valente. Orionplanning: Improving modularization and checking consistency on software architecture. In *3rd IEEE Working Conference on Software Visualization (VISSOFT 2015) – Tool track*, 2015. 8, 127

[SAE+15b] Gustavo Santos, Nicolas Anquetil, Anne Etien, Stephane Ducasse, and Marco Túlio Valente. Recording and replaying system specific, source code transformations. In *15th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'15)*, 2015. 5, 123, 124, 127

[SAE+15c] Gustavo Santos, Nicolas Anquetil, Anne Etien, Stephane Ducasse, and Marco Túlio Valente. System specific, source code transformations. In *31st IEEE International Conference on Software Maintenance and Evolution*, 2015. 5, 123, 124, 127

[SD12] Quinten David Soetens and Serge Demeyer. ChEOPSJ: Change-Based Test Optimization. In *16th European Conference on Software Maintenance and Reengineering*, pages 535–538, 2012. 58

[sis] Structural investigation of software systems (SISSy) tool. `http://sissy.fzi.de/sissy/`. Online; accessed 26 July 2016. 26

[Som00] Ian Sommerville. *Software Engineering*. Addison Wesley, sixth edition, 2000. 1

[SPA+17] Gustavo Santos, Klérisson Paixão, Nicolas Anquetil, Anne Etien, Marcelo Maia, , and Stéphane Ducasse. Recommending source code locations for system specific transformations. In *24th International Conference on Software Analysis, Evolution, and Reengineering*, pages 1–10, 2017. 123, 125, 127

[SRK+09] Santonu Sarkar, Shubha Ramachandran, G. Sathish Kumar, Madhu K. Iyengar, K. Rangarajan, and Saravanan Sivagnanam. Modularization of a large-scale business application: A case study. *IEEE Software*, 26(2):28–35, 2009. 7

[SSL01] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics based refactoring. In *5th European Conference on Software Maintenance and Reengineering*, pages 30–38, 2001. 23

[SSS14] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*, volume 4 of *10*. Morgan Kaufman, The address, 1 edition, 11 2014. 2, 3, 24, 25, 26, 112

[STV16] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of GitHub contributors. In *24th International Symposium on the Foundations of Software Engineering*, pages 1–12, 2016. 24

[SVA14] Gustavo Santos, Marco Túlio Valente, and Nicolas Anquetil. Remodularization analysis using semantic clustering. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 224–33, 2014. 38

[SYA$^+$13] Dag I. K. Sjoberg, Aiko Yamashita, Bente Anda, Audris Mockus, and Tore Dyba. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, 2013. 25

[TPB$^+$15] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In *37th International Conference on Software Engineering*, pages 403–414, 2015. 29

[TSO12] Minh Tu Ton That, S. Sadou, and F. Oquendo. Using architectural patterns to define architectural decisions. In *Conference on Software Architecture and European Conference on Software Architecture*, pages 196–200, 2012. 9

[TVCB12] Ricardo Terra, Marco Tulio Valente, Krzysztof Czarnecki, and Roberto S. Bigonha. Recommending refactorings to reverse software architecture erosion. In *16th European Conference on Software Maintenance and Reengineering, Early Research Achievements Track*, pages 335–340, 2012. 8, 12, 13

[UGDD10] 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. 39

[VCM$^+$13] M. Vakilian, N. Chen, R. Z. Moghaddam, S. Negara, and R. E. Johnson. A compositional paradigm of automating refactorings. In *27th European Conference on Object-Oriented Programming*, pages 527–551, 2013. 2, 21

[VEdM06] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. JunGL: a scripting language for refactoring. In *28th International Conference on Software Engineering*, pages 172–181. ACM Press, 2006. 2, 20

[VKMG09] Stephane Vaucher, Foutse Khomh, Naouel Moha, and Yann-Gael Gueheneuc. Tracking design smells: Lessons from a study of god classes. In *16th Working Conference on Reverse Engineering*, pages 145–154, 2009. 28

[Yam14] Aiko Yamashita. Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data. *Empirical Software Engineering*, 19(4):1111–1143, 2014. 25

[YMNCC04] Annie Ying, Gail Murphy, Raymond Ng, and Mark Chu-Carroll. Predicting source code changes by mining change history. *Transactions on Software Engineering*, 30(9):573–586, 2004. 18

[ZHB11] Min Zhang, Tracy Hall, and Nathan Baddoo. Code bad smells: A review of current knowledge. *Journal of Software Maintenance and Evolution*, 23(3):179–202, 2011. 27

[ZSPK15] 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. 22, 24, 94

Curriculum Vitae

Personal Information

Name:	Gustavo Jansen de Souza Santos
Date of Birth:	April 26, 1989
Place of Birth:	Campina Grande, Brazil
Nationality:	Brazilian

Education

2014-2017:	**Ph.D. in Informatics**
	RMoD, INRIA Lille Nord Europe
	Université de Lille, France
	`http://rmod.inria.fr/`
2012-2014:	**Master of Science in Computer Science**
	Thesis Title: *Remodularization Analysis using Semantic Clustering*
	Federal University of Minas Gerais
	`http://dcc.ufmg.br/`