System Specific, Source Code Transformations

Gustavo Santos, Nicolas Anquetil, Anne Etien, and Stéphane Ducasse
RMoD Team, INRIA Lille Nord Europe
University of Lille, CRIStAL, UMR 9189
Villeneuve d'Ascq, France
{firstname.lastname}@inria.fr

Marco Tulio Valente
Department of Computer Science
Federal University of Minas Gerais
Belo Horizonte, Brazil
mtov@dcc.ufmg.br

***Abstract*—During its lifetime, a software system might undergo a major transformation effort in its structure, for example to migrate to a new architecture or bring some drastic improvements to the system. Particularly in this context, we found evidences that some sequences of code changes are made in a systematic way. These sequences are composed of small code transformations (*e.g.,* create a class, move a method) which are repeatedly applied to groups of related entities (*e.g.,* a class and some of its methods). A typical example consists in the systematic introduction of a Factory design pattern on the classes of a package. We define these sequences as *transformation patterns*. In this paper, we identify examples of transformation patterns in real world software systems and study their properties: (i) they are specific to a system; (ii) they were applied manually; (iii) they were not always applied to all the software entities which could have been transformed; (iv) they were sometimes complex; and (v) they were not always applied in one shot but over several releases. These results suggest that transformation patterns could benefit from automated support in their application. From this study, we propose as future work to develop a macro recorder, a tool with which a developer records a sequence of code transformations and then automatically applies them in other parts of the system as a customizable, large-scale transformation operator.**

***Index Terms*—Software Maintenance; Restructuring; Refactoring Tools; Code Transformation; Rearchitecting.**

I. Introduction

Software systems must evolve to remain pertinent in their context and continue to be used. Evolution can be achieved by activities such as adding new functionalities, to correct bugs or to react to changes in the system's environment. It also sometimes happens that a larger transformation effort is undertaken, for example to migrate the system to a new architecture, to update APIs on which the system depends, or to improve the organization of the system. This large transformation is referred in literature as rearchitecting [3].

In this paper, we studied examples of rearchitecting in real world systems. We found evidences of systematic application of sequences of small code transformations (*e.g.,* create a class, extract a method, remove an attribute). We purposely avoid to use the term "refactoring" because rearchitecting consists in a substantial effort that (i) can profoundly modify the whole system and (ii) often have to break consistency and change behavior in the middle of the process. Moreover, as oppose to previous work on identifying patterns of change [12, 13, 26, 32], transformations have a higher level of granularity and they are specific to the system in which we found them. We present an illustration example of a real pattern in Section III.

The sequences of transformations we found are (i) system specific; (ii) to the best of our knowledge, applied manually; (iii) sometimes not applied to all software entities that should be transformed; (iv) sometimes complex, *i.e.,* including different small transformations in sequence; and (v) applied over several revisions of the system under analysis. We call these sequences of transformations, *transformation patterns*.

From these findings and from evidences in the literature, we claim that transformation patterns might be found in many systems beyond the ones discussed in this paper. Moreover, the manual application of such patterns can be error prone due to the repetition of the changes involved. In our study, we found examples in which developers missed opportunities to apply a pattern, or did not apply all the changes described in the pattern. As a consequence, one would benefit from some automated support in their application. The ways this support could take have been discussed in literature and we provide discussion on a tool to support building custom and reusable transformation patterns at the end of the paper.

The contributions of this paper are: (i) we demonstrate the existence of transformation patterns in real software systems; (ii) we validate these patterns manually; and (iii) we validate the properties that make transformation patterns challenging to apply manually, in order to discuss the importance of their automated support.

This paper is organized as follows: Section II provides an overview of the context of our research and the problem considered. We define transformation patterns in Section III. Section IV presents the investigative study on real-world transformation patterns, which we describe in Section V. Section VI presents the study results. Finally, Section VII presents future work and Section VIII concludes.

II. Related Work

Developers and researchers alike have long perceived the existence of systematic source code transformation operators. This existence led them to propose some automation to reduce the possibility of mistakes and ease the work of developers. As a consequence, Integrated Development Environments (such as Eclipse and VisualWorks) include refactoring operators as an alternative of composite transformations that define very specific, behavior-preserving tasks.[1] Some of these refactor-

[1]Therefore, we use the term *refactoring* in this section, since it is the term authors use in the referenced papers.

ings include renaming an entity (and all its occurrences in the source code) and assisting the creation of methods or variables. They are inspired by the refactoring catalog proposed by Fowler et. al [7].

However, recent work proved that refactoring tools are underused. Murphy-Hill et al. [22] and Negara et al. [23] conducted different studies based on the refactoring tools proposed by ECLIPSE platform. Both studies lead to the conclusion that, when a refactoring operator is available for automated application in ECLIPSE, the developers prefer to perform this operator manually. Vakilian et al. [29] found similar findings 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 composition of small well-known refactoring operators that will produce the same outcome as a complex built-in refactoring operator. These results emphasize the importance of letting the developer having control of the maintenance process.

This lack of trust in automated refactorings points toward a need for the definition of a custom automated refactoring by the developer, which we will discuss in Section II-A. This custom refactoring is defined as an aggregation of small code change operators, which are discussed in Section II-B.

A. Definition of Composite Transformations

The definition of composite program transformations has been proposed in the literature for at least three decades [2, 24]. France et al. [8] propose to transform models by applying design patterns [9]. 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 in [16, 19]. Defining composite transformations for the application of design patterns is very *generic* compared to ours that aims to also solve system specific issues.

Other approaches work with existing source code. Similar to most model approaches, Kozaczynski et al. [15] also proposes composite transformations using application conditions and operators. There are also approaches which are based on code examples [4, 5]. The transformation is defined from an example of the source code before and after the transformation. In such cases, the example is a type of explicit condition. However, in these approaches, the operators are *simple* and consist in code insertion, replacement, and deletion only.

COCCINELLE relies on text matching to define bug patches [17]. 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).

This state of the art shows that: (i) personalizing and having control of the code changes is important for the developer; and (ii) existing approaches enable either to define simple, localized, and eventually specific changes; or more complex but generic transformations. There is a lack of approaches to provide specific, eventually complex, and not localized source code transformations.

B. Change Operators

Javed et al. [11] categorize change operators on source code in three levels, described as follows.

Level one operators are atomic and describe generic elementary tasks. For example, these operators are routinely proposed in ECLIPSE as refactorings (*e.g., Rename Variable*), development helpers (*e.g., Create a Method*), and calculated from source code in the CHANGEDISTILLER tool [6]. These operators are generic in the sense that they are independent of the system, the application domain, and sometimes even of the programming language.

Level two operators are aggregations of level one operators and describe more abstract composite tasks. For example, the *Extract Method* is a composition of several atomic changes (*e.g., Create Method*, *Add Statement*, etc.). These operators depend on the programming language they are based on. However, they are still generic because they can be applied to systems from different domains.

Finally, **level three** operators are aggregations of level one or level two operators, and they are domain specific. This classification relies on two major characteristics, the size of the change operators (atomic versus complex) and the application domain (generic versus domain specific).

III. TRANSFORMATION PATTERN DEFINITION

In this section, we describe more precisely the definition of a transformation pattern. It is worth noting that the term "pattern" comes from repetition of code transformations.[2] Therefore, to better illustrate what kind of pattern we are considering, and not to mistake with design patterns, we present an example of real transformations found on a toy e-commerce system, called MYWEBMARKET [28].

A. Illustrating Example

MYWEBMARKET went to several transformations to improve its structure. In one of these transformations, dependencies to the Hibernate framework were isolated in a new package with the help of a Factory design pattern.[3] The transformations were applied only to the classes which methods depend on HIBERNATE, and they are described as follows.

[2]From Merriam-Webster dictionary, "the regular and repeated way in which something happens or is done" [1].

[3]Therefore, the Factory design pattern is a subset of the transformations done in this example.

PATTERN I
MYWEBMARKET'S TRANSFORMATION PATTERN

Context: Isolating framework dependencies using Factory pattern
Applied to: 7 classes.
Condition: $\exists$ class *C* $\notin$ package *PHib* that depends on Hibernate

1. create an interface *IC'* in *PHib*
2. create a class *C'* in *PHib* implementing *IC'*
3. Create a method "*public C' getC'()*" in the factory *FHib*
4. $\exists$ method *M* in *C*
5. and $\exists$ *S* statements $\in M$ creating the dependence on Hibernate
6. extract statements *S* to a new method *M'* in *C'*
7. replace statements *S* by a call *FHib.getC'().M'()*

For each of the selected classes, the pattern does the following: it creates a corresponding Java interface (line 1) and a special class *C'* implementing it (line 2). Both interface and implementing class are created in a new package *PHib*. The new class is instantiated by a method (line 3) in the Factory class *FHib* created to this effect. Then, for a set of statements in methods in *C* that depends on Hibernate (lines 4–5), this set is moved to another method in the special class *C'* (line 6).The statements are then replaced by a call to the factory to create an instance of the special class and call the method with Hibernate dependency (line 7).

Concretely, each step we previously defined consists in a *transformation operator*. Following the definition of Javed et al. [11], we define a transformation operator as an operator that can be atomic or aggregated (levels one or two). Moreover, a *transformation pattern* is composed of an *application condition* and a *sequence of transformation operators*.

The application condition selects, from all of the code entities in a system, which ones are candidate to change. In MYWEBMARKET example, the condition selects all the classes that depend on Hibernate. On the other hand, the sequence of operators is ordered because transformation operators are dependent from each other [21].

Additionally, each application condition describes one or more *bound variables*. These variables are not directly defined in the pattern, but they are necessary to the application of the transformation operators. In this example, the transformation pattern has three bound variables: (i) the class *C* in the application condition, and (ii) the method *M* and (iii) the set of statements *S*, both defined in the internal condition.

Transformation pattern is very similar to the definition of level three operators of Javed et al. [11]. It differs from their definition because we also consider an application condition, and a transformation pattern impacts several interconnected entities, in contrast with a transformation operator that concerns one or few entities. The condition was inspired from most pattern definition approaches (see Section II). Transformation pattern also differs from the mentioned approaches in the sense that transformation patterns also consider level two operators. Finally, in this paper we do not propose another language to define transformation patterns. We used the notation of Pattern I to better understand the pattern.

B. Properties of Transformation Patterns

We define three important properties of a transformation pattern. These properties highlight the need of study, documentation and automation of the patterns. The properties are described as follows and they are evaluated on real patterns in Section IV.

Frequency denotes the number of occurrences of the pattern. Naturally, following the definition of "pattern", this property is the most evident.

Complexity relies on two attributes. The *number of operators* concerns how many transformation operators have to be repetitively applied. Moreover, the *number of bound variables* concerns how many entities have to be considered in each repetition of the pattern. 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 manually applying the pattern is rather complex, either because of the identification bound variables or because of the number of operators.

Finally, it must be reinforced that, contrary to refactorings, 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 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. Moreover, we do not worry at this point with the automation of these patterns. Up to now we use the definition illustrated in Section III-A to describe the patterns we found in Section V.

IV. EVALUATION

In this section we specify some research questions that need answering to show the interest of working with transformation patterns. We also present the real world systems which are subjects of our study.

A. Research Questions

To assert the interest in studying what we call *transformation patterns*, we need to consider some questions that will be listed here. The next sections will be devoted to answering them on a set of real world systems.

RQ1 *Can we identify instances of transformation patterns in other systems?* This is an obvious question, as the concept should have some generality to be of any interest.

RQ2 *Are transformation pattern applied to all the transformation opportunities?* We intend to investigate whether the patterns are applied to all the entities they are supposed to. This question relates to the *frequency* property.

RQ3 *Are transformation patterns applied accurately?* We aim to analyze whether all the transformation operators that we identified as part of the transformation patterns were performed in each instantiation we found. This question relates to the *complexity* property.

RQ4 *Are transformation patterns applied over several revisions of the system?* This question relates to the *recurrence* property.

We describe our case study as follows. We now present our dataset. Section V answers RQ1 by presenting the patterns we found. Then Section VI deals with research questions RQ2 to RQ4. Note that we will not formalize further our research questions (formal hypothesis) or formally test them. All is required at this stage of the research is proof of existence.

B. Target Systems

The dataset is based on previous work with large restructurings of real software systems [27]. We added to this list systems that have undergone a restructuring effort in our research group. In total, we have four Java systems and five Smalltalk systems, and we list them as follows:

ECLIPSE went through a considerable modularization to integrate the OSGi technology. We studied the user interface plugin, which was separated into five new plugins in the version 3.0 and the followings.

JHOTDRAW is a framework for technical graphics. Its rearchitecting dedicated in specializing the interface of color spaces.

MYWEBMARKET is a toy e-commerce system. Its rearchitecting concerns the application of Factory pattern and it is presented in Section III.

VERVEINEJ is a small parser for static analysis of Java programs based on JDT (ECLIPSE plugin for Java). It went through a small rearchitecting of the AST visitors in the early phase of its development.

PETITSQL is another parser, for SQL. Its rearchitecting focused on correcting API usage of the grammar.

PETITDELPHI is a parser for Delphi that has been enhanced to generate an AST from a tokenized tree. The developers restructured it in order to prune the generated AST nodes.

PACKAGEMANAGER is a package management system for Pharo.[4] Its rearchitecting focused on changing the interface to access package metadata.

VERVEINEJ, PETITSQL, and PETITDELPHI are systems in which one of the authors of this paper participated in the past (we come back on this point in the Threats to Validity in Section V-F). It must be noted that this pattern occurred before our study and it was not in influenced by the current analysis. For all of the aforementioned systems, we found examples of transformation patterns. We must report that we studied two other systems for which we could not identify any patterns matching our definition:

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

TELESCOPE is a visualization framework for Smalltalk. It went through series of refactorings to specialize visualization builders.

[4]http://pharo.org/

Table I summarizes descriptive data about our dataset.

TABLE I
SIZE METRICS OF OUR DATASET. EACH LINE DESCRIBES A REARCHITECTING BETWEEN TWO VERSIONS. METRICS ARE SHOWN IN PAIRS (BEFORE AND AFTER THE REARCHITECTING). THE FIRST FOUR SYSTEMS ARE IN JAVA, THE LAST FIVE ARE IN SMALLTALK. SYSTEM IN ITALICS ARE THOSE FOR WHICH WE DID NOT IDENTIFY TRANSFORMATION PATTERNS MATCHING OUR DEFINITION.

	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
MyWebMarket 0.1 / 1.0	1/3	19/25	1/1
VerveineJ 0.77 / 0.87	2/2	8/7	4/5
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*

Because our method for identifying transformation patterns is purely manual, we do not claim that there are no patterns in GENETICALGORITHMS and TELESCOPE. Extracting transformation patterns from code change 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 can happen. In fact, the existence of patterns in most of the systems in our analysis is not a rare condition. In our study, so far we showed that seven out of the nine systems we studied presented some patterns.

V. THE PATTERNS (RQ1)

This section describes the patterns we identified for the systems under analysis, answering the research question RQ1. We describe them using the format we proposed in Section III, *i.e.,* a condition followed by a list of transformation operators. Due to space constraints, we decided to describe the patterns with the highest complexity (*i.e.,* number of transformation operators) or the highest frequency (*i.e.,* number of occurrences). At the end of the section, we discuss some possible threats to the validity of this study.

A. Eclipse

In ECLIPSE, we identified a pattern related to modularizing the `Action` hierarchy. In order to conform to OSGi architecture, ECLIPSE components are separated as plugins. Most subclasses of `Action` were moved from the `workbench` plugin to the `ide` one. Because of that, the class `WorkbenchMessages` is not accessible in the new plugin. All of the invocations to methods of this class are therefore replaced by invocations to methods of a new class, called `IDEWorkbenchMessages`.

We see in this example a complex transformation pattern. It consists of four operators that impact two classes in each sequence of operators (*e.g., C* and `IDEWorkbenchMessages`). All the more, this pattern was applied to 26 classes which

PATTERN II
ECLIPSE'S (FIRST) TRANSFORMATION PATTERN

Context: Action reorganization and registration
Applied to: 26 classes.
Condition: ∃ class *C* ∈ `ui.workbench`
that extends `jface.Action`

1. move class *C* to plugin `ui.ide`
2. ∃ methods $M \in C$, $M_W \in$ `WorkbenchMessages`
and M invokes M_W
3. add a static method M'_W to `IDEWorkbenchMessages`
4. copy the statements of $M_W()$ to $M'_W()$
5. replace the invocation to M'_W in method M
by an invocation of `IDEWorkbenchMessages.`$M'_W()$

seems a high number enough to justify some effort in automating it. This pattern illustrates a situation that we found often: some transformation patterns have internal conditions. We will come back to this point in Section VII.

In ECLIPSE, we found another transformation pattern related to the use of the `SafeRunnable` abstract class. The pattern consisted in discovering all classes that extend `SafeRunnable` and override the method `handleException()`, and remove these overriding methods. The pattern is not listed here because it is very short (only one operator). However, it was applied to 72 classes in different versions, most of them anonymous classes which are hard to manually inspect. We come back to this discussion in Section VI-C.

B. JHotDraw

The rearchitecting in JHOTDRAW was applied to color spaces hierarchy, which extends AWT. All `ColorSpace` classes must implement a new interface called `NamedColorSpace`, which has only one method, called `getName`. The transformation also includes the use of a Singleton design pattern (lines 3–6). It is worth noting that this pattern impacts not only the class extending `ColorSpace` but also all the classes that instantiate this class (because of the application of the design pattern in lines 5–6). We come back to this pattern in Section VI-B.

PATTERN III
JHOTDRAW'S TRANSFORMATION PATTERN

Context: Defining new interface for color spaces
Applied to: 9 classes.
Condition: ∃ class *C* that extends `ColorSpace`

1. add interface `NamedColorSpace` to *C*
2. add a method `getName()` in *C*
3. add a private attribute `instance` in *C*
4. add a static method `getInstance()` in *C*
5. ∃ method *M* that invokes `new C()`
6. replace `new C()` by call to `C.getInstance()`

C. PetitSQL

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. Before the transformation, this treatment was done by calls to a filtering method on the collection in `ASTNodesParser`.[5]

It was estimated that this implementation made the code hard to understand and it was best to do the filtering individually in each rule of the subclass with an already existing `withoutSeparators()` method. Thus, calls to `withoutSeparators()` are added in the subclass methods, and the use of the filtering methods on collection is removed.

PATTERN IV
PETITSQL'S TRANSFORMATION PATTERN

Context: Fixing Parser API usage
Applied to: 6 methods.
Condition: ∃ method $M_P \in$ `ASTNodesParser`
and ∃ method $M_G \in$ `ASTGrammar`
and M_P invokes M_G
and M_P then invokes `Collection.filter()`

1. override M_G in `ASTNodesParser` (called M'_G here)
2. put a call to `super` in M'_G
3. add a call to `withoutSeparators()` in M'_G
4. remove call to `filter()` in method M_P

This pattern is applied only six times and impacts only one class (*e.g.,* `ASTNodesParser`). However, the pattern has a complex application condition, with two bound variables that depend on each other. Some automation would significantly avoid errors from the developer due to checking the condition manually.

D. PetitDelphi

For each grammar rule defined in `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 whole infrastructure was modified to suppress the creation of the intermediary node (TypeDeclaration). The pattern then removes the method which creates this node in the subclass and the class representing the node (line 1).

PATTERN V
PETITDELPHI'S TRANSFORMATION PATTERN

Context: Prune AST node hierarchy
Applied to: 15 methods.
Condition: ∃ method *M* ∈ `PDDelphiSyntax`
and *M* is a disjunction of other rules

1. remove the method *M* in `PDDelphiParser`
2. ∃ class *C* that *M* instantiates (`new C()`)
3. remove the class *C*

[5]Similar to *Collection2.filter()* of the Google guava library: http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/index.html

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.

E. PackageManager

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. We describe the most frequent pattern in this section.

In this 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 VI
PACKAGEMANAGER'S TRANSFORMATION PATTERN

Context: Correcting package usage
Applied to: 66 classes.
Condition: ∃ class *C* that extends `PackageSpec`
and ∃ method *M* in *C* named 'dependencies'

1. add statement creating an instance *A* of Array
2. ∃ statement *S* in *M* and *M*:
   calls `PackageVersion»addDependency:` with param. *P*
   and calls `Dependency»addConstraint:` in cascade
3. remove *S*
4. create association *E* with parameters *P* and `self»version`
5. create statement adding *E* to *A*
6. add statement returning *A*

The remaining patterns follow the same idea. The *second* pattern decomposes the method `platform` in `PackageSpec` in two methods, changing calls to setter methods to an array with a string (with 19 occurrences). The *third* make similar changes to methods `repositories` of the same hierarchy (64 occurrences). And finally, the *fourth* pattern update calls to setters to class `Dependency` in classes that do not extend `PackageSpec` (7 occurrences).

Summary: We identified instances of transformation patterns in seven out of nine 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 of these systems.

F. Threats to Validity

Some additional points need to be made to clarify the validity of this first investigative study. Considering that we analyzed only one instant of each system's change history (referent to their rearchitecting), identifying transformation patterns in these systems is a very positive result. Moreover, these systems use different programming languages which ensures a wider applicability of our results. On the other hand, four points need to be discussed further.

First, *the authors were among the developers* in three of the systems under analysis. We selected these systems because discovering transformation patterns in unknown system is very difficult. Similar code change mining approaches have the same challenge [31, 32]. The fact that some of us knew three rearchitecting cases was indeed a big help. But this fact does not alter the validity of the results because the analysis of rearchitecting cases was post-mortem. The patterns occurred before this study and our participation in the development only helped us to re-discover them.

Second, *the dataset is over-represented by parsing tools*. This point is related to the previous one because both concern the same three systems. Although three out of nine systems under analysis belongs to parsing domain, not all of the transformation patterns we found are specific to such domain. For example, in VERVEINEJ, the pattern is related to the use of a Visitor design pattern, which can apply to domains other than parsing. In PETITSQL, the pattern is related to better use the API to filter a collection which again is independent of the parsing domain. It does not seem that this point should be a threat to validity. However, it is important to have a more replicable approach to identify transformation patterns in other domains (see also Section VII).

Third, *most of our systems are small*. We acknowledge this is the biggest threat in this study. This point may bias both for and against our general claim that transformation patterns happen and their automation is needed. It biases against our claim because, as exemplified with VERVEINEJ, the less entities a system has, the less occurrences a pattern will present. The pattern in VERVEINEJ showed only three occurrences in two (out of eight) classes of the system. The much larger ECLIPSE has patterns with 26 and 70 occurrences. On the other hand, the point biases in favor to our claim because one might suppose it is easier to systematically apply some transformations in a system when this system is small and well-known (see also Section VI-A). As a first result, the patterns in ECLIPSE and PACKAGEMANAGER seem to indicate that the size of the system is not an issue.

Fourth, there seems to be a *correlation with presence or introduction of known design pattern in our transformation patterns*. This point is exemplified in the very first transformation pattern we presented (MYWEBMARKET, Section III), which introduces the Factory design pattern. Other examples occur in VERVEINEJ (Visitor) and JHOTDRAW (Singleton). Indeed, repeated code modification can be based on the introduction of a design pattern. In these systems, this introduction helped us to identify and describe their transformation patterns. However, we observed that the patterns are not limited to the design pattern definition. For example, only three out of five operators of JHOTDRAW's transformation pattern are concerned with the Singleton design pattern. There are additional modifications which make the transformation pattern more system specific than the design pattern.

VI. Relevance of the Transformation Patterns

Research questions RQ2 to RQ4 are intended to evaluate whether the transformation patterns we identified would have enough relevance to justify extra effort to automate them in some way. We discuss these questions according to the properties we defined in Section III. After that, we discuss some threats to the validity of this evaluation in Section VI-D.

A. Are Transformation Patterns Applied to all the Transformation Opportunities? (RQ2)

The analysis consisted in applying the condition we set for the transformation pattern to the entire system, and count how many entities the condition matched. Then, we count the number of occurrences of the pattern and compare to the preceding value. The expectation is that developers, in lack of special tool to help them, might have forgotten some possible application opportunity. For consistency reasons, more specifically with RQ3, we counted as occurrences cases where the patterns were not accurately applied. Because this condition is defined by ourselves, we will come back to this evaluation in the threats to validity. The results are summarized in Table II.

TABLE II
Number of possible occurrences of the transformation patterns and number of actual occurrences. The number of occurrences in parenthesis is the number after the first revision (See also RQ4)

Transformation patterns	Entities matching *condition*	Pattern occurrences
Eclipse (first)	34	26
Eclipse (second)	86	(70)72
JHotDraw	9	9
MyWebMarket	7	7
PackageManager (first)	66	66
PackageManager (second)	19	19
PackageManager (third)	64	64
PackageManager (fourth)	7	7
PetitDelphi	19	(15)19
PetitSQL	6	6
VerveineJ	3	3
Average	29	27

The patterns in JHotDraw, MyWebMarket, PetitSQL, and VerveineJ were applied in all the opportunities. These are also the patterns with the least frequency. It seems natural that with so little potential occurrences, the developers had less difficulties in identifying them all. However in PackageManager, all the patterns were totally applied and most of them are very frequent. This fact is due to the modification that motivated the pattern. The deletion of setter methods broke the code, therefore the developers had to systematically correct the system to make it run again.

For Eclipse and PetitDelphi, we found some possible occurrences that were not applied initially in the first revision, even for the small PetitDelphi. In this later case, the developer confirmed that the *condition* for the pattern is correct and that they were aware of the missing occurrences at the time of the transformation. However, it was not part of the restructuring effort they were conducting and therefore they chose to leave it for some latter work. They actually ended up applying all the possible occurrences of the pattern. This case relates to RQ4 and we come back on it in Section VI-C.

Summary: The transformation patterns were not always applied to all the opportunities in which the condition matched. 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.

B. Are Transformation Patterns Applied Accurately? (RQ3)

In this section we investigate whether the patterns, when applied, included all the transformation operators. We expect that a pattern may not be accurately applied because it is complex (as defined in Section III). For this matter, Table III summarizes the number of operators and bound variables in the transformation patterns we identified in this study.

TABLE III
Number of transformation operators (as described in Section V) and number of bound variables

Transformation patterns	Number of operators	Number of bound variables
Eclipse (first)	4	3
Eclipse (second)	1	1
JHotDraw	5	2
MyWebMarket	5	3
PackageManager (first)	5	4
PackageManager (second)	9	6
PackageManager (third)	4	4
PackageManager (fourth)	2	2
PetitDelphi	2	2
PetitSQL	4	2
VerveineJ	2	2

Most of the patterns have relatively few bound variables. For example in PackageManager, the variables are very related to each other (*e.g.,* a statement *S* that calls a method *M* with a parameter *P*). This pattern has five operators.[6] Overall, we found that most of the patterns were accurately applied. This was obvious for patterns such as the second of Eclipse.

However, the patterns were not consistently applied in Eclipse (first) and JHotDraw. Based on Table III, these patterns were not the most complex of the dataset. Therefore, it does not seem to exist a relationship between their complexity and the fact that some of their operators were not applied. Specifically for the number of operators, we cannot draw the same conclusion because operators can be level two operators.

The first pattern in Eclipse was applied to 26 classes in version 3.0 (see Pattern II), so line 1 of the pattern was always applied. However, one class (`SelectionListenerAction`) did not apply the lines 3 to 5. We checked the code and found that this class did not have invocation to `WorkbenchMessages` in the first place. Therefore, the internal condition of the pattern (line 2) is not met and the rest of the pattern is ignored. For this reason, we consider that this pattern was applied entirely.

[6] See Pattern VI, as defined in line 1 and lines 3 to 6.

For JHOTDRAW (see Pattern III), none of the changed classes implement the Singleton design pattern accurately (lines 3 to 6). All of them missed changing the constructor accessibility to `private`. Because of that, there are still direct instance creations (`new`) to three of these classes. This fact means that both the *design pattern* (Singleton) and the *transformation pattern* were not applied accurately.

Still in JHOTDRAW, we also found that two classes do not implement the Singleton pattern at all. These classes do not have the unique instance of the class (Pattern III, line 3) and the method that returns this instance (line 4). Moreover, one more class does not extend `NamedColorSpace` (line 1). There is no instantiation of these three classes in the project. It is possible that these classes are not considered as "active" and should be removed from the system in the future.

Summary: In some of the patterns, not all of their operators were applied. It does not seem to exist a correlation between this fact and the complexity metrics we proposed.

C. Are Transformation Patterns Applied over Several Revisions? (RQ4)

Finally, we investigate whether the patterns were applied over several releases. Except for ECLIPSE and PETITDELPHI, all of the patterns were applied in one revision. As discussed in Section VI-A, (i) the patterns with least frequency were applied in one revision, which is expected; and (ii) the patterns in PACKAGEMANAGER had to be performed at once because previous modification introduced error in the code.

The first pattern in ECLIPSE was initially applied in 26 classes at the first revision. This pattern consisted in moving `Action` classes to another component and replacing invocations to a new class of this component (see Pattern II). Between versions 3.2 and 3.3, a total of 28 `Action` classes were added in the `ide` component. Naturally, these classes did not have to replace invocations to the class modified by the pattern (line 5) because they invoke this class directly. We did not succeeded in obtaining a condition for this continuous addition. This addition shows that the result of this pattern continued to be observed even when the pattern itself was not applied anymore.

Table IV describes the revisions under analysis in ECLIPSE and PETITDELPHI. For each revision, we count the number of entities that applied the pattern, accurately (see RQ3) or not.

TABLE IV
SELECTED REVISIONS OF TWO SYSTEMS WITH THE NUMBER OF OCCURRENCES OF A TRANSFORMATION PATTERN FOR EACH

System	#Rev.	Date	Occurrences
Eclipse (second)	3.0	06/25/04, 12:08	70
	3.1	06/27/05, 14:35	71
	3.2	06/29/06, 19:05	72
	3.3	06/25/07, 15:00	72
	3.7	06/13/11, 17:36	72
PetitDelphi	210	11/19/14, 14:52	15
	211	11/19/14, 18:56	17
	212	11/26/14, 18:17	18
	213	12/03/14, 18:23	18
	214	12/22/14, 15:55	19

For both systems, the transformation patterns took around five revisions to be applied. This fact in itself already confirms our research question. The revisions in PETITDELPHI extended over one month, in which its restructuring effort demanded four hours per week. The second pattern in ECLIPSE consists in a single operator (remove an overriding method). Although modern IDE's like ECLIPSE have facilities that would allow one to discover all the possible candidates for applying the transformation patterns (*e.g.,* searching all references to the `SafeRunnable` class), it took two years to go from 70 to 72 occurrences, and after seven years, there were still 14 ($86-72$, see also RQ2) opportunities left.

Summary: Some transformation patterns were applied in not one but several revisions.

D. Threats to Validity

Some points need to be discussed about the validity of this study on transformation patterns. The evaluation of RQ2 depends on the condition attached to the patterns and whether this condition covers entities that were not actually transformed. Because in many cases, the condition was identified by us, there is always a risk that we overlooked some detail and that the condition we defined is too extensive.

In PETITDELPHI for example, the condition was not clear until the author who participated in the development helped to define it correctly. Even so, we have for this system, concrete example of not applying immediately all the opportunities, because the developers wanted to focus on a specific part of the system at a given moment. For ECLIPSE (second pattern), there are also cases in which the pattern was not applied initially, and the developers came back to it later.

In RQ3, we found very little occurrences of a transformation pattern not accurately applied. Similarly, in two out of three cases in RQ4, the patterns were either slightly recurrent or not recurrent at all. This result might be caused by characteristics of these system specific patterns or a consequence of our identification methodology. Because we had to reverse engineer the patterns from changes, our attention was obviously drawn to the more regular patterns with more occurrences. Therefore, because of this setting, it is possible that we just did not identified actual patterns that were not accurately applied.

VII. FUTURE WORK

We showed that transformation patterns actually appear in real world systems. They can be very simple or more complex, with few or many occurrences. In this section, we consider what could be done to help developers in using them.

A. Transformation Pattern Identification

In this study, the patterns were identified manually. This process was done in three steps. First, we *automatically compute changes* between two versions of a rearchitecting in term of level one operators (*e.g.,* added, deleted, or modified lines). This process was done using the diff calculator provided by ECLIPSE for Java systems and the TORCH tool [10] for Smalltalk systems. Second, we *manually analyze* the set of

transformations to identify repeating groups that could be the seed for candidate transformation pattern. For example in ECLIPSE (second), we found that several methods with the same name were removed. And third, we *identify the condition* by inferring common properties in the entities we found in the previous step. In the same example, we noticed that all the changed methods belonged to classes in the same hierarchy.

This process is tedious due to the huge size of the list of changes, and it is error prone because the condition might relate to a wide range of properties of the changed entities. There is existing work on mining similar code transformations to discover patterns from these transformations [12, 14, 20, 25, 31, 32]. However, none of these approaches seems yet to be able to discover patterns matching our definition. Specifically, they lack identifying an application condition in order to check other application opportunities. As discussed in Section V, identifying the right condition is very complex. In some cases, we had to ask the developers to understand and define the right condition. Some automated support in this context is too complex and it is therefore discarded for future work.

B. Transformation Pattern Definition

As identified in this paper, the developers already applied the transformation patterns manually. Therefore, the definition of these patterns might be useful to identify other possible application opportunities. Moreover, one might envision a scenario in which the developer perceives the repetitive sequences of actions during the rearchitecting process. In both cases, a clever tool would offer the developer support to define a pattern and apply it total or semi automatically.

We discussed in Section II different approaches to define and apply transformation patterns. First, approaches based on code examples take one or few examples of code modification and extract patterns from them. However, these approaches rely on simple and localized operators. We identified patterns which constituting operators are more general and involve different code entities at once. Second, approaches based on Domain Specific Language (DSL) rely on a language to express both conditions and transformation operators [18, 30]. However, the language limits the set of transformation operators to compose. Moreover, the complexity for the developer is increased (*i.e.,* to learn another language), whereas rearchitecting is an occasional and already time-consuming process.

We introduce the idea of a "macro recorder", a tool which would allow the developer to (i) *record* a sequence of transformations while they are applied a first time, either manually or with the assistance of refactoring tools; (ii) *store* and parameterize the transformations to allow the generalization of the pattern (see next section); and (iii) *apply* automatically the sequence of transformations afterwards on different entities. For the last step, the developer could explicitly point to the entities to transform or specify an application condition.

Such a tool would have three main contributions. First, a transformation pattern is build by gradual composition of smaller transformations. The developer knows what each recorded transformation is doing. This contribution addresses the trust of such automated support. Second, the developer configures the pattern upfront. As opposed to code example approaches, the recorder knows the exact operators and in what order they need to be applied. And third, the approach does not force the developer to know a limited set of transformation operators, as opposed to DSL approaches. Manual, level one operators could also be supported.

C. Pattern Parameterization

In our study we observed that most patterns have few bound variables. Moreover, most of these variables are related to each other, *i.e.,* a method which belongs to a class. This is a good property as it would simplify the work of explaining to an automated tool how to apply a pattern systematically.

On the other hand, we observed cases in which the sequence of operators depend on properties that are not easy to express in a condition. For example, in PACKAGEMANAGER one of the operators depends on the result of a call to a given method (see Pattern VI, line 4). In PETITDELPHI, the developers removed one class (see Pattern V, line 3) based on part of the name of method that was removed before (in line 1). We extracted a different condition based on dependencies between these entities, and this condition had the same outcome as the previous one. In this context, we also propose to define parameterizable patterns in order to allow variations in the operators they include and the entities they modify.

VIII. CONCLUSION

Maintenance is an important activity in the software system life cycle. It can take several forms such as bug correction, adding new functionalities or more occasionally substantial modification of the architecture. During a large maintenance effort, developers can perform repetitive changes, sometimes complex either by the number of operators involved or by the condition to select the right entities to change.

In this paper, we defined *transformation pattern* as an application condition and an ordered sequence of source code transformation operators. We studied several cases of punctual efforts to rearchitecture real world systems and we showed that many of them contained some transformation patterns. The evaluation leads to the conclusion that patterns are really used in the rearchitecting process. They are language independent (we studied Java and Smalltalk systems) but system specific.

We also showed that transformation patterns were not always completely applied on all the entities that should be transformed, their constituting operators were not always entirely applied, and patterns were applied over an extended period of time and several revisions. To identify these patterns proved to be complex, either to identify the sequence of operators or to define the correct application condition.

By providing automated support to apply the pattern, the developers would avoid errors due to their repetitive application and their complexity. We end this paper by discussing an idea of a tool to define transformation patterns based on reuse of manual transformations. We proposed some research paths for future work on this issue.

REFERENCES

[1] Definition of Pattern by Merriam-Webster dictionary. http://www.merriam-webster.com/dictionary/pattern. Accessed: 2015-06-30.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[8] 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.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.

[10] Veronica Uquillas Gomez, Stephane Ducasse, and Theo D'Hondt. Visually supporting source code changes integration: The torch dashboard. In *17th Working Conference on Reverse Engineering*, pages 55–64, 2010.

[11] 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.

[12] 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.

[13] Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *31st International Conference on Software Engineering*, pages 309–319, 2009.

[14] 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.

[15] Wojtek Kozaczynski, Jim Ning, and Andre Engberts. Program concept recognition and transformation. *IEEE Transactions on Software Engineering*, 18(12):1065–1075, 1992.

[16] 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.

[17] 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.

[18] 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.

[19] Slavisa Markovic and Thomas Baar. Refactoring OCL annotated UML class diagrams. *Software and System Modeling*, 7(1):25–47, 2008.

[20] 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.

[21] Tom Mens, Gabriele Taentzer, and Olga Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, 6(3):269–285, 2007.

[22] 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.

[23] 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.

[24] James M. Neighbors. The draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, 10(5):564–574, 1984.

[25] Kai Pan, Sunghun Kim, and E. James Whitehead Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[26] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. Template-based reconstruction of complex refactorings. In *26th International Conference on Software Maintenance*, pages 1–10, 2010.

[27] Gustavo Santos, Marco Tulio Valente, and Nicolas Anquetil. Remodularization analysis using semantic clustering. In *Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 224–233, 2014.

[28] 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*, pages 335–340, 2012.

[29] Mohsen Vakilian, Nicholas Chen, Roshanak Zilouchian Moghaddam, Stas Negara, and Ralph E. Johnson. A compositional paradigm of automating refactorings. In *27th European Conference on Object-Oriented Programming*, pages 527–551, 2013.

[30] 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.

[31] Annie T. T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9): 574–586, 2004.

[32] 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.