A New Architecture Reconciling Refactorings and Transformations

Balša Šarenac[a], Nicolas Anquetil[b], Stéphane Ducasse[b], Pablo Tesone[b]

[a] *University of Novi Sad, Faculty of Technical Sciences, Trg Dositeja Obradovića 6, 21102 Novi Sad, Serbia*
[b] *University Lille, Inria, CNRS, Centrale Lille, UMR 9189 - CRIStAL, F-59000 Lille, France*

Abstract

Refactorings are behavior-preserving code transformations. They are a recommended software development practice and are now a standard feature in modern IDEs. There are however many situations where developers need to perform mere *transformations* (non-behavior-preserving) or to mix refactorings and transformations. Little work exists on the analysis of transformations *implementation*, how refactorings could be composed of smaller, reusable, parts (simple transformations or other refactorings), and, conversely, how transformations could be *reused* in isolation or to compose new refactorings. In a previous article, we started to analyze the seminal implementation of refactorings as proposed in the PhD of D. Roberts, and whose evolution is available in the Pharo IDE. We identified a dichotomy between the class hierarchy of *refactorings* (56 classes) and that of *transformations* (70 classes). We also noted that there are different kinds of preconditions for different purposes (applicability preconditions or behavior-preserving preconditions). In this article, we go further by proposing a new architecture that: (i) supports two important scenarios (interactive use or scripting, i.e., batch use); (ii) defines a clear API unifying refactorings and transformations; (iii) expresses refactorings as decorators over transformations, and; (iv) formalizes the uses of the different kinds of preconditions, thus supporting better user feedback. We are in the process of migrating the existing Pharo refactorings to this new architecture. Current results show that elementary transformations such as the Add Method transformation is reused in 24 refactorings and 11 other transformations; and the Remove Method transformation is reused in 11 refactorings and 7 other transformations.

Keywords:. Refactorings, transformations, preconditions, source code transformation, behavior preservation

1. Introduction

Refactorings are behavior-preserving code transformations. The seminal work of Opdyke [1] and the Refactorings Browser (the first implementation of refactorings by Roberts and Brant [2, 3, 4, 5]) paved the way to the spread of refactorings [6]. They are now a standard feature in modern IDEs [7, 8, 9, 10, 11]. A lot of research has been done on refactorings such as for their detection [12], missed application opportunities [13, 14], practitioner use [7, 8, 9, 10], their definition [15, 16, 17, 18, 19], or atomic refactorings for live environments [20]. Several publications focus on scripting refactorings [21, 22, 23, 24, 25]. Finally, some work has attempted to speed up existing refactoring engines, as for Java [26].

Still, from a daily development perspective, refactorings and their behavior-preserving forms are not enough [15, 27, 28]. Non-behavior-preserving code transformations are also needed [18, 19, 29]. For example, consider replacing all the invocations of a given message with another one (which we call ReplaceMessageSend(msg1,msg2)). ReplaceMessageSend is not equivalent to RenameMethod: the former requires msg2 to exist, whereas the latter does not require it to exist. Also, the former (ReplaceMessageSend) does not need to deal with possible overriding implementations of msg1 whereas the refactoring must rename them too.

ReplaceMessageSend should just update all the msg1 invocations to msg2 invocations. Such a transformation will typically not preserve behavior, yet it is a need that arises in real development situations. It is clear that ReplaceMessageSend has similarities with the RenameMethod refactoring, but it would be awkward [1] to perform it by applying RenameMethod only. When in need of such a source code transformation, a developer is left to perform the changes manually or with a code rewriting engine that can be cumbersome to use [28].

Defining some specific code transformations such as ReplaceMessageSend and letting the Pharo developers define their own transformations are our long-term engineering goals. In this paper, we explore a new refactoring engine architecture to do so. Note that our goal is not to provide an out-of-the-box, language-independent, solution for these problems. However, we believe that the architecture presented in this article can be used to model other dynamically-typed object-oriented languages such as Ruby, Python, Javascript, Lua with an effort to handle specific language semantics adaptation. Refactoring engines must be language-specific and sometimes even editor-specific. Nevertheless, in the discussions, we will try to highlight the parts that are specific to our situation, so that readers can reuse our ideas.

We worked on the seminal implementation of refactorings as specified in the Ph.D. of D. Roberts [5] and available in the Pharo IDE [30]. To make the discussions clear, we name:

[1] The developer would need to copy msg2 in a paste buffer; then remove it before executing the rename refactoring; then rename manually (without refactoring) msg2 back into msg1; and finally, paste back the copied method to its original definition!

- **Legacy version**: The version of the Refactoring Browser that has been available in Smalltalk and Pharo up to Pharo version 11. In this paper and the code, *legacy version* classes are prefixed with “RB”. This implementation follows mostly the descriptions [2, 3, 4] and the specification in the Ph.D. of D. Robert [5]. There are some small differences that do not impact our work. For example, the PhD uses postconditions while the implementation does not.

- **New architecture**: The version presented in this article, whose implementation is available in Pharo 12, features classes that are prefixed with 'Re'. It is important to note that not all classes in the legacy refactoring engine have yet been migrated to the new architecture. Therefore, in the new architecture some classes are still from the old architecture and are prefixed with “RB”, whereas the ones that have been migrated are prefixed with “Re”.

In the legacy version (Pharo 11), the inheritance hierarchy of *refactorings* contains 56 classes, and the one of *transformations*, 70 classes. The implementation does not allow for simple reuse of refactorings in different conditions (namely non-behavior-preserving transformation of the source code). It is not possible to combine different refactorings to achieve a higher-level evolution of the code [28].

Based on this situation, we defined a new architecture that supports two scenarios: *interactive* use and *scripting* (batch) use of refactorings. This new architecture defines a clear interplay between *refactorings* and *transformations* by offering a compatible API. It simplifies refactoring implementation by defining them as decorators over transformations. In such an architecture, transformations have applicability preconditions and refactorings have behavior-preserving preconditions [31]. The reification of conditions (preconditions' elementary components) also supports better user feedback. All such changes improve the reuse of transformations and refactorings as well as the precondition logic.

This article revisits the duality of refactorings and transformations, and how they can be reused together [29, 31]. Although it may seem trivial to state that refactorings can be decomposed in preconditions + transformations, it is not. Our work highlights that there are different kinds of preconditions and our new architecture clearly separates applicability preconditions and behavior-preserving preconditions.

Our contributions are the following:

- A new *architecture supporting interactive and scripting* uses of refactorings and transformations. This new architecture is based on *driver objects* that support the interactive application of refactorings.

- The unification of the refactoring and transformation APIs for simpler use and ease of combination;

- The definition of refactorings as decorators on transformations improving their reuse;
- The separation of the different kinds of preconditions (applicability preconditions in transformations, behavior-preserving preconditions in refactorings) improves their reuse.

We are well aware that Pharo Refactoring Browser (*legacy version*) is only one refactoring engine. It has evolved since 1996 in the hands of multiple developers introducing new refactorings and reorganizing the code. Just as the Java refactoring engine [26], it has shortcomings. The paper does not aim to criticize one particular implementation of the refactoring engine but rather identify possible issues in this implementation to infer more generic rules.

The outline of the paper is the following: Section 2 sets the vocabulary used in the paper and illustrates the need for a more flexible and versatile refactoring engine from a user (i.e., developers) point of view. It makes the case for supporting two important usage scenarios, interactive and scripting, and it stresses the need for transformations and refactorings. Section 3 details the *legacy version* implementation of the refactoring engine, highlighting its shortcomings regarding the needs expressed before. Section 4 presents the *new architecture* which introduces *drivers*; unification of the refactorings and transformations APIs; and reification of pre-conditions. Each of these new concepts is presented in turn. Section 5 presents a first evaluation of the *new architecture* that shows that it supports better reuse of precondition, refactoring, and transformation logic. Section 6 discusses the process of migrating to the *new architecture* and some of its benefits. The paper closes with the related work discussion (Section 7) and the conclusions (Section 8).

2. The need for a more flexible and versatile refactoring engine

In this section, we define the domain of our study: We look at source code modifications from a user (i.e. developer) point of view, to highlight the need for both refactorings (behavior-preserving modifications) and, transformations, behavior-agnostic modifications. We discuss also the need to share as much logic as possible between the two. By *behavior agnostic*, we mean that the modification of the source code has no knowledge of, and does not care about, the behavior of the code.

We start by defining the vocabulary that is used in this paper, then we show that there is a need for both refactorings and transformations, as well as a need to reuse their logic and support different usage scenarios.

2.1. Definitions

We first clarify the vocabulary used in this paper.

Refactoring: behavior preserving modification of the source code. Refactorings were introduced by Opdyke [1] and first specified and implemented in Smalltalk by Roberts and Brant [2, 3, 4]. Then they were intensively studied as shown in Section 7.

Transformation: behavior agnostic modification of the source code. This is a modification of the source code without consideration for the impact on its behavior. Transformations should, however, not be *syntax agnostic* or *semantic agnostic*, which means, they should take care of producing source code that is syntactically correct (it parses) and semantically correct (it compiles);

Precondition: Typically, the implementation of *refactorings* includes some *preconditions* that may check the possibility of applying the refactoring. For example, the refactoring RENAME METHOD(oldName, newName) first checks that an oldName method exists and, that a newName method does not already exist;

Applicability precondition: A precondition checking that a refactoring can be applied, *i.e.,* that the *transformation* can be applied (independently of behavior preservation) [31]. An applicability precondition may be checking, for example, that an entity targeted by the refactoring, exists, or that given information (such as names) is correct [15].

Behavior-preserving precondition: A precondition checking whether the application of refactoring would break the system once applied [31]. For example, the REMOVE CLASS refactoring checks that the class is not referenced anymore, that it does not have subclasses, or that it is not a metaclass.

Note that the differentiation of the two types of preconditions is important because transformations do not care about program behavior and therefore do not need behavior-preserving preconditions while refactorings may use both types.

2.2. Different user needs

During software development, the need for both refactoring and transformation becomes apparent. We will illustrate this with the ADD METHOD refactoring/transformation.

There is also a need for different usage scenarios: interactive and scripting (batch).

Refactorings vs. transformations. Consider the example of a tool to assist in adding a method:

- As a refactoring, ADD METHOD checks that the user is not overriding a locally defined method or an existing method in the superclass.
- As a transformation, it should be able to create a new method only checking that the name is valid and that the class exists.

With only the refactoring, users wouldn't be able to override existing methods since that override might be a non-behavior-preserving change. On the other hand, with only the transformation, users would always be able to override methods from the superclass which could introduce bugs.

This example points to the concrete need for both refactoring and transformation, where users can choose the right behavior in their working context. Section 3.2 proposes another similar example with Replace Message Send refactoring/transformation.

Interaction vs. Scripting.. Most refactorings require additional information to be gathered ("parameters") for their execution. For example, for renaming a method, the refactoring needs the class holding the method to rename, the existing method and, the new name.

Refactorings are often applied interactively during development sessions and the needed information is naturally gathered through interactions with the user: RenameMethod prompts the user to give a new name or whether arguments should be permutated. With that information, it checks that the provided name is valid and continues its execution. A good interaction, however, is often more than just a single prompt. It should also give adequate feedback and adapt to different scenarios: As another example, removing a class should propose different interaction scenarios such as (1) browsing the references to the class, (2) folding the class state and methods in its subclasses, (3) simply removing the class, (4) removing the class and browse its users,...

Literature has highlighted the need for developers to trust a refactoring engine. For this, they need to have a complete understanding of what is going on at each step. Thus, the quality of the interaction is key to the acceptance and use of refactorings. The current work stems from perceived shortcomings in the way the *legacy version* was dealing with some less common scenarios.

But refactorings can also be used in script (batch mode) without user interaction [21, 22, 24, 25]. In scripting mode, refactorings will be fully configured by the calling script. This does not preclude them from checking if the provided information is valid before execution.

We can see that these two modes share most of the logic of the refactoring (validation and execution), but their flows are different. In particular in interactive mode, preconditions can be decomposed to support the user flow (*e.g.,* prompting and validating inputs). Refactorings and transformations should be designed in a way that enables easy reuse of the logic between these two modes.

There is a two-level reuse need: first between transformations and refactorings and second between the two uses (interaction and scripting).

2.3. Reuse of logic

A frequent source code modification is to change the name of an invoked method. This can happen either by renaming the method (same behavior, new name); changing the invoked method (different behavior, new method); or, adding or removing parameters to a method (similar behavior, different calling conventions).

Whatever the case, the modification will need to change all invocations of the old name to invocations of the new name. It must also check the number of parameters[2] and their order (which may have been altered). In the case of renaming a method, and adding/removing parameters, the existing method must also be modified, and additional checks will be needed, for example, if it is overridden in subclasses, or it overrides a super-class method.

This logic is complex, and requiring to navigate and manipulating an abstract representation of the source code and considering all possible special cases and their implications. From a software engineering point of view, it is important to be able to reuse some parts of the logic. Such reuse is, however, also questioned in the presence of duplicated behavior in transformations and stressed by the definition of new generation refactorings, such as the atomic refactorings supporting live object programming [20].

2.4. Research questions

We want to understand whether refactorings can be implemented in terms of transformations that would be independent operations, usable by the developers for scripting some development actions but also usable in interactive sessions. To support the duality of refactorings and transformations both at a conceptual and implementation level, this article wants to answer the following questions.

- Can refactorings and transformations share their logic? Which parts can be reused? How much can be reused between them?
- What is the architecture to support the two usage scenarios (scripting and interactive session)?
- What is the API exposed for each scenario?

3. Legacy implementation

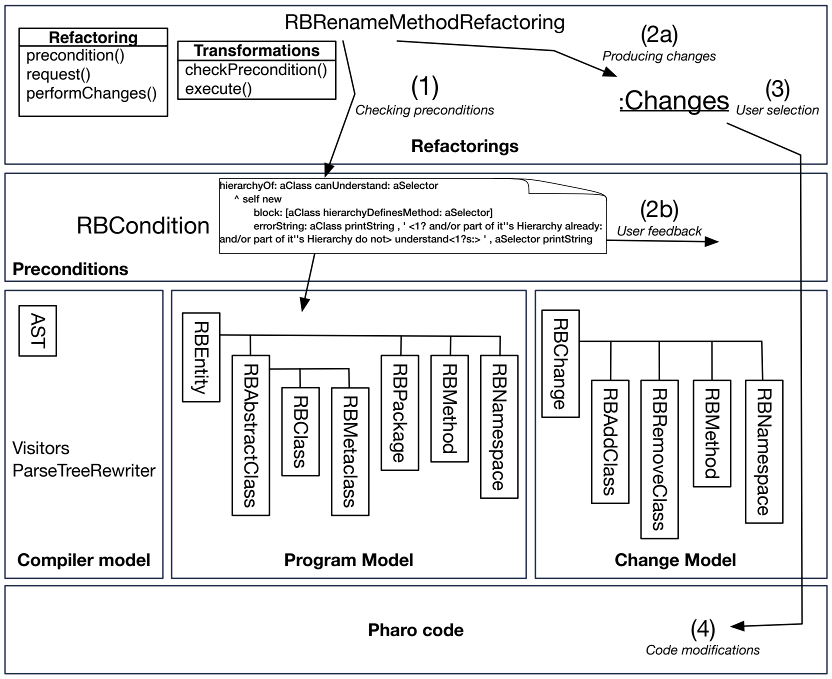
We now present the *legacy version* of the refactoring engine and highlight some of its shortcomings concerning the research questions exposed above. This section contains material that is specific to Pharo and its refactoring engine, but the critical analysis in the last subsection presents conclusions that apply outside of this restrained scope.

3.1. The legacy Refactoring Browser architecture

The work presented in this article takes its roots in the implementation of refactorings as done by J. Brant and D. Roberts [2, 3, 4] and their evolution as available in Pharo [30]. Since multiple developers maintained and evolved the original code, our analysis will report a situation that is not one described in the original document. It may happen that some preconditions are missing,

[2]In Pharo, parameters are not statically typed, so type verification is not required.

or were changed, or that new refactorings are not extending existing ones. Appendix Appendix A presents the list of original refactorings, as described in the Ph.D. of D. Roberts [5]. The Pharo implementation contains more refactorings, as shown in Appendix Appendix B, it also contains transformations (see Appendix Appendix C).



```
hierarchyOf: aClass canUnderstand: aSelector
    ^ self new
        block: [aClass hierarchyDefinesMethod: aSelector]
        errorString: aClass printString , ' <1? and/or part of it''s Hierarchy already:
and/or part of it''s Hierarchy do not> understand<1?s:> ' , aSelector printString
```

Figure 1: Overview of the legacy refactoring engine architecture on one example: First, refactorings check their preconditions using the precondition library (RBCondition) (step 1). If a precondition fails, the user is notified (step 2b), otherwise the refactoring produces a list of change objects (step 2a) expressed through the Change Model. The list of changes is proposed to the developer (step 3) for validation or selection before being applied, modifying the actual code (step 4).

In the *legacy engine*, refactorings and transformations are defined in separate hierarchies [28] with different APIs. A refactoring uses a **program model** to check **preconditions** (*step 1, Figure 1*) and produce code changes. The preconditions notify the users in case of violation (step 2b in the figure). The output of a refactoring is a sequence of changes (2a Fig. 1) that, once applied to the existing code, will perform the refactoring. Users can select the changes that will be applied (step 3). It results in the modification of the actual Pharo code (step 4).

The infrastructure of the legacy engine is:

The **program to be refactored** (bottom row). Pharo runtime is composed of objects (classes, compiled methods, ...) representing the program and libraries as well as their execution [32].

The **compiler** model (3rd row, left) consists of a basic AST (concrete syntax tree with information about the characters placement to be able to reproduce the source code) and companion visitors. A parse tree rewriter is available and used by the refactoring engine.

The **program** model (3rd row, center). It is a representation of the program entities (represented as instances of RBMethod, RBClass classes) and their AST (for methods). The code model is simple and traditional: A class knows its package, superclass, subclasses, instance variables, and methods. The ASTs are reified on demand from a given method entities [30, 33]. The engine does not directly use the reflective language API [32] to perform preconditions and refactorings, it uses its own program model.

The **change** model (3rd row, right). Change objects describe the actions that will be performed on the actual code. Change objects represent all the operations required to modify the code or metaobjects (since Pharo is a reflective system [32]) of the program: addition/removal/rename of class, method, fields, and method source code modifications.

The **precondition library** (2nd row). Using the program model, the class RBCondition defines a large set of predicates as static methods that are used in the refactoring preconditions. The conditions raise exceptions and sometimes request information from the user.

The **refactoring definitions** (top row). This component contains the refactoring and transformation definitions. Every refactoring action is performed on the program model elements. Preconditions are expressed using condition predicates and the refactoring produces change objects. It should be noted that, in the legacy implementation, such definitions are entangled with user interactions (such as getting parameters, new method name request,. . .).

Changes are not automatically applied: the developer is prompted with a list of changes (acting as a preview of the refactoring of the code) among which developers can select the ones to be applied. In addition, offering the developer a preview of a refactoring application in terms of changes acts as a kind of transaction. The system is only modified when the developer accepts the proposed changes, and developers may cancel some or all of the proposed changes. The refactoring engine lets developers select a portion of the proposed changes. This step is useful because, without variable static typing, it may be difficult to identify the invocation of one method with a very common name. This selection actually means the user may turn a refactoring into a non-behavior-preserving modification. This is not seen as an issue, because (1) the users see what the refactoring proposes to do and understand what is happening, and (2) this gives them additional flexibility on the source code modification.

3.2. Reuse implementation: the case of ChangeMethodName

We saw in Section 2.3 an example of reuse needs to implement a source code modification like changing the name of an invoked method. We identified four cases where such a change might be used: (i) renaming the method — same behavior, new name —; (ii) changing the invoked method — different behavior, new method —; or, (iii) adding or (iv) removing parameters to a method — similar behavior, different calling convention.

In the legacy version, these actions are implemented by four subclasses of the abstract class RBChangeMethodNameRefactoring which we describe in the following.

RBRenameMethodRefactoring.. This class implements the refactoring that changes a method's name in the target class and all invocations (message sends) in the senders of the method. This is done by inheriting (and not overriding) the transform method from RBChangeMethodNameRefactoring that (see listing 1):

- renames the implementors with the new name (line 2);
- then renames all the old references to the method (line 3);
- and finally removes the old selector (line 4).

```
1  RBChangeMethodNameRefactoring >> transform
2    self renameImplementors.
3    self renameMessageSends.
4    self removeRenamedImplementors
```

Listing 1: RBRenameMethodRefactoring behavior, inherited from RBChangeMethodNameRefactoring.

The method transform and these three steps are all implemented in the abstract super-class and inherited in RBRenameMethodRefactoring.

RBReplaceMethodRefactoring.. The class RBReplaceMethodRefactoring[3] implements another related action: the transformation that *replaces* the invocations of a method by invocations to another method but does not affect the previously invoked method itself. This class also inherits from the abstract RBChangeMethodNameRefactoring, but it overrides its transform method (see Listing 2).

[3]The name is misleading as this is a transformation and not a refactoring: changing the method invoked cannot guarantee behavior preservation.

```
RBReplaceMessageSendTransformation >> transform
  self replaceInAllClasses
    ifTrue: [ self renameMessageSends ]
    ifFalse: [ self renameMessageSendsIn: {class} ]
```

Listing 2: The transformation of the class RBReplaceMethodRefactoring.

RBAdd/RemoveParameterRefactoring.. The two last subclasses are dealing with method parameters. RBAddParameterRefactoring verifies that the new method is correct (does not shadow an existing temporary variable, does not shadow existing methods) and it ensures that callers of the original method are updated with a default value specified by the developer. RBRemoveParameterRefactoring mainly checks that the new method does not override the existing one and that there is no reference to the removed parameter.

In this implementation, reuse is based on inheritance for the refactorings and transformation. In the following section, we present another reuse mechanism used in the library of preconditions.

3.3. Legacy precondition implementation

Preconditions are central to the expression of refactorings and in this section, we analyze the *legacy implementation.*

The preconditions are expressed as:

- simple (low-level) conditions implemented in class-side methods of the RBCondition class;
- composition of these simple conditions using two classes: RBConjunctiveCondition and RBNegationCondition;
- methods implemented in the program model (see Figure 1).

Example 1.. The following precondition method (Listing 3) checks that a class effectively defines a variable (lines 3 and 4) before creating its accessors. It uses two methods from RBCondition: definesClassVariable:in: and definesInstanceVariable:in:.

```
1 RBCreateAccessorsForVariableRefactoring >> preconditions
2   ^ classVariable
3     ifTrue: [ RBCondition definesClassVariable: variableName asSymbol in: class ]
4     ifFalse: [ RBCondition definesInstanceVariable: variableName in: class ]
```

Listing 3: A basic case of preconditions using methods from RBCondition.

Example 2.. Listing 4 shows another precondition example: it directly uses methods from the program model: RBAbstractClass¿¿hierarchyDefinesInstanceVariable: (line 3). It checks, before pulling up an instance variable, that it exists in all the subclasses (see Listing 4).

```
RBPullUpInstanceVariableRefactoring >> preconditions
  ^RBCondition withBlock:
    [ (class hierarchyDefinesInstanceVariable: variableName)
      ifFalse: [ self refactoringFailure: 'No subclass defines ' , variableName ].
    (class subclasses
      anySatisfy: [ :each | (each directlyDefinesInstanceVariable: variableName) not ])
      ifTrue: [ self
        refactoringWarning: 'Not all subclasses have an instance variable named.<n>
         Do you want to pull up this variable anyway?' , variableName , '.' ].
    true ]
```

Listing 4: Pull Up Instance Variable refactoring preconditions.

Example 3.. The Remove Class refactoring checks behavior preservation by implementing its own precondition methods (lines 8 to 11), that call simpler methods from RBCondition (see Listing 5).

```
1   RBRemoveClassRefactoring >> preconditions
2
3     ^ classNames inject: self emptyCondition into: [ :sum :each |
4         | aClassOrTrait |
5         aClassOrTrait := self model classNamed: each asSymbol.
6         aClassOrTrait ifNil: [
7           self refactoringFailure: 'No such class or trait' ].
8         sum & ((self preconditionIsNotMetaclass: aClassOrTrait)
9          & (self preconditionHasNoReferences: each)
10         & (self preconditionEmptyOrHasNoSubclasses: aClassOrTrait)
11         & (self preconditionHasNoUsers: aClassOrTrait)) ]
```

Listing 5: Remove Class preconditions.

3.4. Critical analysis of the legacy implementation

The implementation described above has some interesting features, but we believe there are also shortcomings that should be addressed. This analysis can also serve as a requirement list for a more generic refactoring engine implementation.

Positive points.

- *Separated program model.* The engine does not directly use the reflective language API [32] but its own program model. This allows it to refactor code that does not need to be executable in the current environment itself.
- *Efficient precondition checking.* The program model also supports fast validation of preconditions and fast execution of refactorings. According to Kim *et al.,* [26], the Java refactoring engine is slow because it does not have a model of programs other than AST.

Negative points.

- *Blurry precondition families.* A previous analysis [31] identified different families of preconditions. We agree with this important distinction by sorting preconditions as *applicability preconditions* or *behavior-preserving preconditions.* The legacy implementation does not make this distinction. Yet, it is important because refactorings and transformations do not have the same needs in terms of preconditions.

- *Duplication of logic.* Transformations got added on the side of refactorings [28]. While the intention to be able to support transformations in addition to refactorings was good, the realization was problematic since transformations could be configured to act as refactorings leading to a duplication of behavior, preconditions, and code. It was unclear how transformations could fully replace the refactorings.

- Bad separation of concerns is witnessed by the REMOVE CLASS refactoring. It checks behavior preservation by implementing its own precondition methods, which call a simpler method from RBCondition. This hampers the reuse of such behavior by other refactorings.

- *Mixing user interactions and precondition checking.* The definitions of the refactorings are entangled with user interactions (for example, see end of Listing 4) which hampers logic reuse (reuse of preconditions, refactorings, and transformations). These user interactions are based on exception signaling. This situation makes scripting and interactive application of refactorings difficult to achieve.

- *Difficulties identifying precondition violations.* The legacy implementation partly reifies preconditions with RBAbstractCondition and its three subclasses (RBCondition, RBConjunctiveCondition, RBNegationCondition). Preconditions are implemented as static methods of RBCondition.

  While the definition of the precondition predicates as simple methods is working, this reification is limited because it can only test whether the precondition holds or not. Finding what program elements violate the predicates to report the problem to the user, requires re-implementing a query similar to the precondition logic. It makes the definition of the interactions with the user, full of exception handling and code duplication.

- *Cumbersome user feedback.* Reusing and combining preconditions currently relies on the two classes (RBConjunctiveCondition, RBNegationCondition). Such a combination, while based on a nice design, leads to totally unclear, cumbersome feedback to the users, especially when the precondition is violated, which sometimes leaves the user with no idea about the actual reason for the precondition failure.

- *Lack of systematic reuse.* The boundaries between program model API and refactorings/transformations are unclear. The program model is the lowest API on which preconditions are expressed and on which the elementary source code modifications are performed. Many refactorings are using this API to perform checks and code transformations. When an elementary refactoring uses only one elementary operation it acts as a reification of this operation and as such can be reused by other refactoring instead of forcing each refactoring to duplicate the same precondition checks.

4. New architecture and implementation

In this section, we describe the new architecture we designed. It supports the reuse of logic as well as the two usage scenarios: interactive and scripting modes. We discuss the relationship between refactorings and transformations and how the formers reuse the behavior of the latter. We also discuss the implementation of interactive versus scripting refactoring.

4.1. Overview of the new architecture

The new architecture we designed is illustrated in Figure 2. Compared to the legacy architecture (Figure 1), the different responsibilities that were part of the old refactorings are now split into several objects: *Driver* for interactions with the user (interactive mode); *Transformation* for applicability preconditions and modification of the code; *Refactoring* for expressing behavior-preserving preconditions and extra code changes that would be required to preserve behavior; and *Conditions* for expressing the preconditions, checking them and register the possible violations. This architecture reuses the *program* and *change* models already existing in the legacy implementation (See Section 3.1).

- **Interaction Drivers** (top row, left). Interaction drivers are objects responsible for requesting information from the user, configuring refactorings, and finally executing them. An interaction object can launch different refactorings depending on the choices of the user. Subclasses of InteractionDriver encapsulates the logic for a family of refactorings. For example, removing a class may lead to pushing down class state in subclasses (refactoring Remove Class Pushing State To Subclasses) or when the class is empty just reparenting subclasses to the superclass (refactoring Remove Class). They have the responsibility to configure and use a configured refactoring to check step-by-step preconditions (*e.g.,* if a given name is correct). In particular, they use the result of reified condition objects to report to the user situations where the refactoring cannot be executed due to a precondition violation.

- **Refactorings and Transformations** (top row, right). Transformations define applicability preconditions expressed as reified preconditions. Refactorings act as decorators over transformations ensuring that the transformation does not alter the behavior. Both produce change objects that are validated by the user. These change objects implement a kind of transaction (in the database sense).

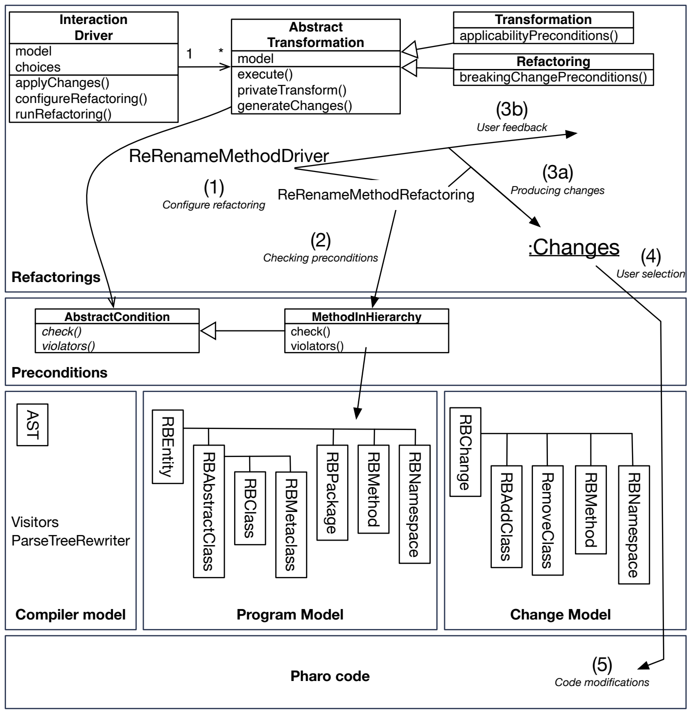


Figure 2: Overview of the new refactoring engine architecture on one example: First, a driver creates and configures a refactoring (1), and then the driver asks the refactoring to check its preconditions (2). In case of violation, the driver reports to the user (3b). When preconditions hold, it asks the refactoring to produce changes (3a). The driver then previews the changes to the user (4) who can select which ones to apply. Finally, the changes are performed on the code (5).

- **Reified conditions** (2nd row). Reified conditions are objects that check a condition by searching the program model on which the refactoring or transformation is applied. In addition to indicating whether the condition is met or not, they differ from the legacy implementation (with precondition methods) in that they can also store information about the objects (classes, methods, etc.) that would cause a precondition to fail. We call these objects *violators*, they are important to provide developers with appropriate feedback, either in the event of a refactoring or transformation failure or to assist developers in reconfiguring the refactoring to ensure its success (see Section 4.4).

A new interaction flow.. As shown in Figure 2, a driver creates and configures a refactoring (*Step 1*). It then requests the refactoring to execute its preconditions (*Step 2*). If the preconditions fail the driver reports to the user (*Step 3b*) as opposed to the precondition itself reporting in the legacy implementation (see Figure 1). When the preconditions hold, the driver asks refactoring for the changes and presents them to the developer (*Step 3a*). The developer can select changes (*Step 4*) and finally, the driver applies them (*Step 5*). In this new implementation, the driver is in charge of the execution flow, whereas it was the refactoring itself that did that in the legacy implementation.

4.2. Polymorphic API

In the legacy implementation, refactorings and transformations weren't designed to be used interchangeably. They were part of different class hierarchies, and even though their execution flows had similar steps, their implementations differed significantly. To enable the interplay of refactorings and transformations, the need for a new API arose.

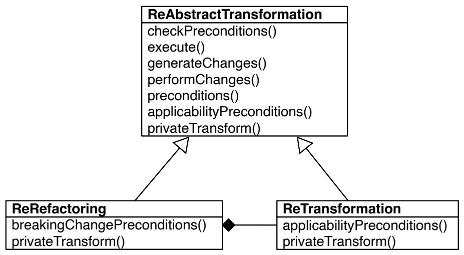


Figure 3: Class diagram representing new refactoring and transformation design.

The new API was designed so that refactorings and transformations can easily be composed, as well as used interchangeably in scripting or interactive execution. The classes ReRefactoring and ReTransformation are part of the same hierarchy, and share the common superclass ReAbstractTransformation (see Figure 3). Thus, they share the same API for checking preconditions and performing program model transformations.

The new API is structured in three layers:

- The first layer is execute method (Listing 6) that will execute complete transformation/refactoring without any user interaction. This method is used in scripting mode.
- The second layer is generateChanges (Listing 7) and performChanges (which are the steps of execute method). This layer is used in interactive mode where the user needs to confirm the changes before performing them on the image.
- The third layer is checkPreconditions and privateTransform (which are the steps of generateChanges). This layer is used in interactive mode when there is a need for more fine-grained control over execution (*e.g.,* user confirmation is needed for continuation).

Layering the API like this enabled the reuse of the whole execution logic between interactive and scripting modes, and additionally, it enabled the interaction driver to have more fine-grained control over the execution flow.

We will now briefly examine methods from these layers.

```
1  ReAbstractTransformation >> execute
2    self generateChanges.
3    self performChanges
```

Listing 6: The execute methods of ReAbstractTransformation.

The execute method (Listing 6) is used during scripting mode, where no user interaction is required. Developers configure the refactoring and invoke the execute method. The method generates changes by invoking generateChanges (line 2) described below. Changes are objects from the change model and performChanges (line 3) applies them to the source code. As a result, the program is modified.

```
1  ReAbstractTransformation >> generateChanges
2    self checkPreconditions.
3    self privateTransform
```

Listing 7: The generateChanges methods of ReAbstractTransformation.

Creating changes for refactorings and transformations (Listing 7) is done in two steps: first, check the preconditions (line 2), then generate the code change actions (line 3). Each actual refactoring/transformation defines its own preconditions. They are retrieved by the checkPreconditions method and executed. If a precondition is not satisfied, it raises an error by default. The error can be "caught" in interactive mode, to give the user the possibility to correct some parameters (see Section 4.4).

In the second step, the privateTransform method is invoked. At the level of the superclass ReAbstractTransformation, it is an abstract method that is therefore redefined in every refactoring and transformation to implement their own program model transformations.

Finally, in the third layer, there is an API for retrieving preconditions:

- preconditions: to get all preconditions of refactoring or transformation;
- applicabilityPreconditions: to get applicability preconditions; and,
- breakingChangePreconditions: for behavior-preserving preconditions. This last one is implemented only by ReRefactoring as transformations do not have behavior-preserving preconditions.

Additionally, some individual reified preconditions (*i.e.,* instances of an AbstractCondition subclass) can be obtained with specific methods (*e.g.,*preconditionHasNoReferences).

4.3. Transformations and refactorings interplay

Anquetil *et al.,* [31] showed that there are different types of preconditions. Those types are described in sections 2.1 and 3.3. Using such distinctions was an insight for us as it helped clarify the difference between refactorings and transformations.

Legacy situation.. In the legacy implementation, this distinction of the preconditions did not exist. Both transformations and refactorings had preconditions. The difference was that transformations didn't check their preconditions by default. They had to be explicitly configured to do so by the developer. This behavior was introduced to support the composition of transformations because the composition has an impact on whether and when all preconditions of the composed transformations should be checked[4]. One may need to check all preconditions upfront or on the contrary, the precondition of a second transformation would only be fulfilled after applying a first transformation.

Unfortunately, this behavior leaving the choice of applying or not the preconditions to the user could also be a source of bugs when the user calls a transformation without remembering to check its preconditions.

New implementation of Transformations.. In the new architecture, we propose that transformations do not have *behavior-preserving preconditions*, but only *applicability* preconditions (see Figure 1). Since transformations are behavior-agnostic modifications of the source code, it doesn't make sense for them to have behavior-preserving preconditions. To avoid execution errors, it is, however, crucial for them to perform applicability precondition checks. Therefore, the new default is that a *transformation* implements its *applicability* preconditions. Without this, in scripting mode, a user could try to perform a transformation to add an accessor to a field that doesn't exist. These cases should never happen, since such actions would break the program and cause errors.

[4]We do not deal with the composition of transformations in this article.

New implementation of Refactorings.. Refactorings on the other hand need to have both behavior-preserving preconditions and applicability preconditions (see Figure 1). So refactorings perform the same actions as the transformations (source code modification and applicability preconditions checks) and have additional checks for behavior-preserving preconditions and additional code changes to ensure behavior preservation. This corresponds to the *decorator* design pattern [34]. One can view refactorings as decorators over transformations that ensure behavior preservation while leaving the bulk of the source code modification to the transformation. This is possible because we unified the API of refactorings and transformations (see Section 4.2).

Therefore, the new generic implementation of refactorings is to have behavior-preserving precondition and delegate applicability precondition implementation and execution logic to an "inner" transformation (or several ones). Additionally, refactorings can decorate some pre- or post-execution logic to ensure behavior preservation changes in the code.

4.4. Interaction drivers

In interactive mode, refactorings are executed using commands invoked either through GUI buttons or shortcuts. A command creates a new instance of an interaction driver (for the given refactoring) and supplies it with any information it has at hand. For example, when right-clicking on a method to call the Remove Method refactoring, the command can provide the name and class of the method to remove. After instantiation, the command invokes runRefactoring which is the main method of the interaction drivers.

```
 1  ReRemoveClassInteractionDriver >> runRefactoring
 2    self configureRefactoring.
 3    refactoring checkApplicabilityPreconditions.
 4
 5    haveNoReferences := refactoring preconditionHaveNoReferences.
 6    emptyClasses := refactoring preconditionEmptyClasses.
 7    noSubclasses := refactoring preconditionHaveNoSubclasses.
 8
 9    haveNoReferences check & emptyClasses check & noSubclasses check
10        ifTrue: [ self removeClassChanges ]
11        ifFalse: [ self handleBreakingChanges ]
```

Listing 8: The runRefactoring methods of ReRemoveClassInteractionDriver.

Listing 8 shows the runRefactoring method of the ReRemoveClassInteractionDriver. It first configures the refactoring (line 2), which usually means creating an instance of the refactoring class and possibly inquiring for additional information needed. For example in a Rename refactoring, it would involve asking for the new name.

Next, runRefactoring delegates to its internal refactoring to check its applicability preconditions (line 3). The applicability preconditions will raise an error and stop execution if one of them fails. After that, the interaction driver checks all behavior-preserving preconditions (line 5 to 7). If one of the behavior-preserving preconditions fails, the handleBreakingChanges method is invoked (line 11), this method is shown in Listing 9. If, on the other hand, all of preconditions pass driver will execute the default refactoring, which is in this case ReRemoveClassRefactoring.

```
1  ReRemoveClassInteractionDriver >> handleBreakingChanges
2    | select items |
3    items := OrderedCollection new.
4    items add: (RBRemoveClassReparentChoice new
5                 driver: self;
6                 classesHaveSubclasses: noSubclasses isFalse;
7                 emptyClasses: emptyClasses isTrue).
8    (noSubclasses isFalse and: [ emptyClasses isFalse ]) ifTrue: [
9      items add:
10       (RBRemoveClassAndPushStateToSubclassChoice new driver: self) ].
11   haveNoReferences isFalse ifTrue: [
12     items add: (RBBrowseClassReferencesChoice new driver: self) ].
13   select := SpSelectDialog new
14             title: 'There are potential breaking changes!';
15             label: self labelBasedOnBreakingChanges;
16             items: items;
17             display: [ :each | each description ];
18             displayIcon: [ :each |
19               self iconNamed: each systemIconName ];
20             openModal.
21   select ifNotNil: [ select action ]
```

Listing 9: The handleBreakingChanges method of ReRemoveClassInteractionDriver.

The method handleBreakingChanges is responsible for interacting with the users to take possible actions based on failed behavior-preserving preconditions. It first creates a menu with all possible actions (lines 3 to 12) before displaying it to the users (lines 13 to 20), and finally executing the choice (line 21). When trying to remove a class, the possible actions are:

- The use of the refactoring Remove Class and Reparent Subclasses (lines 4 to 7) is the default option that is always available. This transformation removes the class and if it has subclasses it changes their superclass to be the superclass of the removed class. This option is always shown because we don't want to limit the users with what they can do. We cannot guarantee behavior preservation, but the users might want to perform this anyway, because they know it does preserve behavior, or they do not care.

- The next choice that is included is the refactoring Remove Class and Push State to Subclasses (line 10). This choice is included if the class being removed is not empty (has at least one method or one instance or class side variable) and it has subclasses. In that situation, we might be able to achieve behavior preservation if we push all the state (all variables and methods) to the subclasses.

- The third choice is not a refactoring, but a browse operation (line 12). This choice is shown if the class has references, then we cannot guarantee that the refactor will not break the system, so we offer the user a closer look at the class references.

Supporting reuse via fragments of preconditions.. The method handleBreakingChanges illustrates that the preconditions of the refactorings are used by the driver in a fragmented way (*i.e.,* it calls preconditionHaveNoReferences, preconditionEmptyClasses, and preconditionHaveNoSubclasses). Contrary to the script mode where the engine executes all the preconditions at once calling the method preconditions, here we see that the driver needs a finer decomposition to better support the interaction with the user. The driver uses individual preconditions to show different choices to the user. Based on the noSubclasses condition and isNotEmpty condition driver shows the choice to push the state to subclasses. Based on the haveNoReferences condition driver shows the "Browse references" choice.

This decomposition of preconditions supports the two usage scenarios as well as the reuse of precondition logic.

Partial refactoring instantiation.. In interactive mode, gathering and checking all required information might require partially instantiating a refactoring with the information at hand before getting and checking more information. This is part of the driver's responsibility.

5. Evaluation

In this section, we present some details that validate the pertinence of our new implementation. The process of migrating all the old 56 refactorings and 70 transformations is a long-term endeavor. The bulk of the initial work consisted of defining the new architecture and validating it on some refactorings.

5.1. Evaluation summary

In a nutshell here are the results:

- All the transformations are now only performing applicability preconditions.

- All the legacy refactorings have been migrated to clearly identify applicability or breaking preconditions.

- We migrated two refactorings (Add Method and Remove Method) to the decorator pattern.

- We also implemented 14 drivers of refactorings:

Deprecate Class	Deprecate Method	Rename Class
Rename Instance Variable	Rename Shared Variable	Rename Method
Push Up Method	Push Down Method	Remove Class
Remove Method	Remove Instance Variables	
Remove Shared Variables	Move Methods To Class Side	
Push Down Method In Some Classes		

- We introduced two explicit composite refactorings ReCompositeRefactoring and ReUpFrontPreconditionCompositeRefactoring. The refactoring Remove Instance Variables and Remove Shared Variables take advantage of them to manage a list of elements. More work is needed to take full advantage of them. In addition, we will revise and extend the 18 transformations that are expressed as explicit compositions of transformations as defined by De Santos [28].

In order to assess the level of reuse in terms of preconditions and refactorings, we analyzed all the refactorings and produced the table reported in the Appendix Appendix D. The table shows that 49 refactorings directly define applicability preconditions, 31 directly define behavior-preserving preconditions, 4 refactorings are indirectly reusing applicability and 9 of them are reusing behavior-preserving preconditions via an explicit and manual composition of other refactorings. A summary of the results is presented in the table 1. It is important to note that our future development effort will reinforce such reuse, as we are reusing refactorings instead of duplicating their logic. It should be noted that these numbers do not include the transformations where we also see reuse opportunities.

Table 1: Precondition reuse via explicit refactoring use.

Refactorings ...	
defining applicability preconditions	49
composed of other refactorings that have applicability preconditions	4
without applicability preconditions	3
defining behavior preserving preconditions	31
composed of other refactorings that have behavior preserving preconditions	9
without behavior preserving preconditions	16

In the rest of this section, we demonstrate how we achieved several of the benefits that were our goals: the reuse of precondition logic; the reuse of transformation logic; the composition of refactorings from transformations; and the use of transformations instead of program model "primitives". We then show how a refactoring is defined by manually composing other transformations. Finally, we present some simple composition operators that we defined and started to use. This section concludes with a discussion of the reuse between scripting and the interactive API for refactoring applications.

5.2. Reuse of precondition logic

The refactoring Remove Method case gives an example of how precondition logic is reused.

```
1  ReRemoveMethodRefactoring >> applicabilityPreconditions
2      ^ transformations applicabilityPreconditions
3
4  ReRemoveMethodRefactoring >> breakingChangePreconditions
5      ^ (RBCondition withBlock: [ self checkSuperMethods ])
6      & (RBCondition withBlock: [ self senders isEmpty ]
7          errorString: 'Cannot remove method because it has senders')
8
9  ReRemoveMethodRefactoring >> preconditions
10     ^ self applicabilityPreconditions & self breakingChangePreconditions
```

Listing 10: Remove Method refactoring precondition checking logic.

Listing 10 shows the applicabilityPreconditions method of ReRemoveMethodRefactoring. It uses the applicabilityPreconditions of its decorated transformation (line 2), and it defines its own behavior-preserving precondition checks (line 4). We decided that it is more logical to check applicability preconditions before behavior-preserving preconditions.

5.3. Reuse of transformation logic

Once more, the class ReRemoveMethodRefactoring is employed to illustrate the reuse of transformation logic.

```
1  ReRemoveMethodRefactoring >> privateTransform
2    transformation privateTransform
3
4  ReRemoveMethodTransformation >> privateTransform
5    self definingClass removeMethod: selector
```

Listing 11: Reusing transformation logic.

The reuse of this logic is shown in the Listing 11. Both refactoring and transformation have a privateTransform method that is part of their unified API. This method in ReRemoveMethodRefactoring delegates to the ReRemoveMethodTransformation, which decorates the execution of privateTransform (line 2). The aforementioned method then performs the actual change by removing the selector from its parent class (line 5).

5.4. Composition of refactorings from transformations

We now turn to the case of the Replace Message Sends and Rename Method refactorings.

In Section 3.2, we discussed their old implementation: they were part of the same hierarchy, inheriting from the abstract ReChangeMethodNameRefactoring that implemented significant portion of the required behavior. One downside of that implementation was that the entire hierarchy was tightly coupled.

The new design favors composition over inheritance. The behavior that was previously inherited is now reused through the appropriate transformations (Figure 4): ChangeMethodNameRefactoring uses ReplaceMessageSendTransformation to update all the method invocations (Listing 12, lines 7 to 14). The latter is an independent transformation, not a subclass of ReChangeMethodNameRefactoring. ReChangeMethodNameRefactoring uses ReRemoveMethodTransformation to remove renamed implementors (Listing 13, line 7).

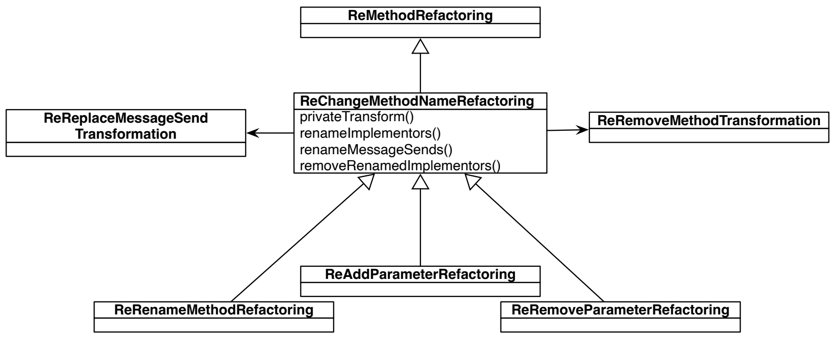

ReMethodRefactoring
ReChangeMethodNameRefactoring
privateTransform()
renameImplementors()
renameMessageSends()
removeRenamedImplementors()
ReReplaceMessageSend Transformation
ReRemoveMethodTransformation
ReAddParameterRefactoring
ReRenameMethodRefactoring
ReRemoveParameterRefactoring

Figure 4: Class diagram representing new ReChangeMethodNameRefactoring implementation.

```
1   ReChangeMethodNameRefactoring >> privateTransform
2     self renameImplementors.
3     self renameMessageSends.
4     self removeRenamedImplementors
5
6   ReChangeMethodNameRefactoring >> renameMessageSends
7     self generateChangesFor: (ReReplaceMessageSendTransformation
8           model: self model
9           replaceMethod: oldSelector
10          in: class
11          to: newSelector
12          permutation: permutation
13          inAllClasses: true
14          newArgs: self newArgs)
15          removeRenamedImplementors
```

Listing 12: ReRenameMethodRefactoring reuses ReChangeMethodNameRefactoring.

5.5. Transformations instead of program model primitives

Returning to the case of the Remove Method refactoring, we now examine the transformation.

Listing 13 (line 7) shows the use of Remove Method Transformation in the removeRenamedImplementors method of ReChangeMethodNameRefactoring. The previous implementation directly accessed the program model, which was a duplication of the transformation (however small). Now that transformations and refactorings can be easily composed, it is now possible to invoke the transformation instead of using the program model directly.

```
ReChangeMethodNameRefactoring >> removeRenamedImplementors
  oldSelector = newSelector
    ifTrue: [ ^ self ].
  self implementors
    do: [ :each |
      self generateChangesFor:
        (ReRemoveMethodTransformation selector: oldSelector from: each) ]
```

Listing 13: ReChangeMethodNameRefactoring invokes Remove Method transformation.

Polymorphic API enabled easier composition of refactorings and transformations, making it possible to rely on composition instead of directly using the program model API. This is beneficial because transformations and refactorings have applicability preconditions that ensure nothing syntactically incorrect will be executed. When using the program model API directly, we do not have that certainty unless preconditions are in place to check it. If there are preconditions in place, they would be mere copies of the preconditions of existing refactorings and transformations.

We were able to refactor 24 usages of the program model API in refactorings and 11 in transformations to use ReAddMethodTransformation. Similarly, we reused the ReRemoveMethodTransformation in 12 places in refactorings and 6 in transformations. Likewise, the ReRemoveMethodRefactoring was also reused in 2 other refactorings.

5.6. Extract Method Refactoring as an example of composition

In this section, we show what we call *manual* refactoring/transformation composition. By this, we mean that developers reuse refactorings or transformations to define new ones without using predefined composition operators. We present the implementation of the extract method refactoring. This refactoring is one of the most complex because it has to deal with the visibility of temporary variables and return statements.

This refactoring is invoked by the user with a range of code selection. The applicability preconditions validate if the selected code range can be extracted. These checks include: ensuring that the selection is not part of a cascaded message, that the selected code does not contain any temporaries or arguments that are read before being written, that the selected code does not contain a return statement, and that there is only one assignment in the selected code. For the sake of space, we will not describe each precondition in detail. Instead, we will focus on the manual composition.

Listing 14 shows the buildTransformations method. This method is part of the ReCompositeTransformation API and should return an ordered list of transformations that need to be executed. The refactoring first creates a new method (an instance of RBMethodNode) that will be added to the class. Then it searches for a method with an equivalent tree as the newly created method in the class hierarchy. If such a method exists, the extraction is not needed since we can reuse the existing method and a basic code rewrite is sufficient. The rewrite will change the method's body to use messageSend to the existing method, instead of creating a new method out of the selected code.

When a method with an equivalent tree does not exist, we need to create a new method with the selected code. The method buildTransformationFor: is responsible for returning a list of transformations needed to perform the extraction. This method first creates a new message send, a string that represents a call to the extracted method. After the computation is done, the method returns a list of transformations that:

- Add a new method to the class,
- Replace subtree transformation replaces the selected code with a call (message send) to a previously created method,
- Remove all temporary variables that can be removed.

```
ReExtractMethodTransformation >> buildTransformations
  | newMethodName existingMethod checker messageSend |
  newMethodName := self newMethodName.
  newMethod := self generateNewMethodWith: newMethodName.
  checker := EquivalentTreeChecker new
                model: model;
                on: class;
                extractedFromSelector: selector.
  existingMethod := checker findEquivalentTreeFor: newMethod.
  existingMethod ifNil: [ ^ self buildTransformationFor: newMethodName ].

  messageSend := self messageSendWith: existingMethod ast.
  ^ OrderedCollection with:
         (RBReplaceSubtreeTransformation
           model: self model
           replace: sourceCode
           to: messageSend
           inMethod: selector
           inClass: class)

ReExtractMethodTranfsformation >> buildTransformationFor: newMethodName
  | messageSend |
  messageSend := self messageSendWith: newMethodName.

  ^ OrderedCollection new
      add: (RBAddMethodTransformation
           model: self model
           sourceCode: newMethod newSource
           in: class
           withProtocol: Protocol unclassified);
      add: (RBReplaceSubtreeTransformation
           model: self model
           replace: sourceCode
           to: messageSend
           inMethod: selector
           inClass: class);
      add: (ReRemoveUnusedTemporaryVariableRefactoring
           model: self model
           inMethod: selector
           inClass: class name);
      yourself
```

Listing 14: A manual composite: ReExtractMethodTransformation.

5.7. *Towards first composite operators*

We started to define a basic composite refactoring operator. The default operator is ReCompositeRefactoring. Its semantics is mainly to loop over a list of refactorings.

```
ReCompositeRefactoring >> privateTransform
   refactorings do: [ :each | each generateChanges ]
```

Listing 15: A simple composite: ReCompositeRefactoring.

The second composite operator is simple yet useful: it evaluates all the preconditions of a sequence of refactorings before executing them all. It assumes that none of the refactorings in the sequence modifies the context in which preconditions are verified. Such an operator is used to create a refactoring that removes several instance variables at the same time as shown in Listing 16.

```
ReRemoveInstanceVariableDriver >> configureRefactoring
   refactoring := ReUpFrontPreconditionCheckingCompositeRefactoring new
         model: model;
         refactorings: (variables collect: [:each |
         ReRemoveInstanceVariableRefactoring model: model remove: each from: class]);
         yourself.
   refactoring prepareForInteractiveMode
```

Listing 16: UpFrontPreconditionCheckingCompositeRefactoring usage.

```
ReUpFrontPreconditionCheckingCompositeRefactoring>>
   applicabilityPreconditions
        "Return the list of the all the applicabilityPreconditions of the composite"
        ^ refactorings collect: [ :each |  each applicabilityPreconditions ]

   breakingChangePreconditions
        "Return the list of the all the breakingChangePreconditions of the composite"
        ^ refactorings collect: [ :each | each breakingChangePreconditions ]

   privateTransform
        "pay attention we are not checking preconditions of children at this level"
        refactorings do: [ :each | each privateTransform ]

   violators
        ^ self breakingChangePreconditions flatCollect: [ :cond | cond violators ]
```

Listing 17: ReUpFrontPreconditionCheckingCompositeRefactoring logic.

We also defined ReCompositeContinuingRefactoring: it iterates over a list of refactorings and does not stop at the first failing precondition.

```
ReCompositeContinuingRefactoring >> privateTransform
  refactorings do: [ :each | [ each generateChanges] on: RBRefactoringError do: [ :ex | ] ]
```

Listing 18: ReCompositeContinuingRefactoring logic.

This is one of our future goals to define and assess more complex operators in the same vein as the ones defined by Li and Thompson [22].

5.8. Reuse between two modes: scripting and interactive

The polymorphic API described in Section 4.2 enables reuse between interactive and scripting modes. Interaction drivers use the fine-grained control that the new API allows to:

- invoke custom UI to gather user input when needed
- validate user input
- display warnings for the failed behavior-preserving preconditions
- offer users different actions based on failed behavior-preserving preconditions
- preview refactoring changes before actually performing them

With this API, the driver can plug in any custom UI code it needs in between execution steps. For example, the driver can show a dialog with all the changes that will be performed on the image before it invokes performChanges, or it can check behavior-preserving preconditions and, if they fail, show a warning to the user that the action it is performing might not be a refactoring, but a transformation.

6. Discussion

6.1. On the process

The migration described in the previous sections is not trivial. It requires a substantial refactoring of the legacy code. Here is a list of the tasks:

- Separation of applicability and behavior-preserving preconditions.
- Removal of behavior-preserving preconditions from transformations.
- Reification of individual preconditions.
- Restructuring of refactorings to be decorators of transformations.
- Design of the API for fine-grained interaction between a driver and the refactorings.
- Creation of drivers for refactorings and migrating UI logic to them.

This process is extremely time-consuming as it requires case-by-case analysis. Just the first step of separating applicability and behavior-preserving preconditions demanded a significant amount of time. It implied analyzing each precondition on its own, examining its influence on the refactoring/transformations, and evaluating its potential impact on other preconditions.

Additionally, creating interaction drivers requires a thorough analysis of all the possible paths the user might take when refactoring something. Besides the happy path (*i.e.,* when all preconditions are satisfied), there is a need to take into account all the actions the user can perform if a behavior-preserving precondition fails, as well as which transformation can be performed in that situation and any other non-transforming operation like browsing for senders, implementors, or references.

Our intention is to document this process and our analytical insights in a refactoring catalog. This catalog is envisioned to serve as a valuable resource aiding in the development of a state-of-the-art refactoring engine in modern IDEs.

6.2. On code readability

Our experience working on this large effort for over a year shows that separating transformations and refactorings enables easier semantic changes and improves readability. It helped us identify mistakes, such as whether ReGenerateAccessors should use the ReAddMethod refactoring or transformation.

When developers working on refactorings see that an operation is composed of transformations, they know that no breaking changes are checked. Similarly, when an operation is composed of refactorings, they immediately know that breaking changes must be satisfied for that operation to be executed. This is an important aid for maintaining and enhancing of the code base.

7. Related work

Some researchers used different algorithms, such as multi-objectives [35] or hill-climbing to identify where refactorings could be applied. This is out of the scope of the article's focus. Other researchers worked on model transformations [36, 37], however, the inherent constraints make the work not applicable to our case. This is why we do not include them in this related work.

There is a limited amount of research focused on the engineering and definitions of refactorings themselves, and we focus on them.

Systematic literature surveys.. The authors of [38, 39, 40, 41] present some systematic literature surveys. Little is said about the reuse of refactoring or transformation logic.

Refactoring specification.. Schaffer *et al.,* [42] propose to use dependencies and language extensions [17] to represent refactorings. They use dependencies instead of preconditions. The authors use this representation to specify RENAMEMETHOD in Java [43] as well as correct refactorings for concurrent Java code [44]. They also introduce the notion of *micro refactorings*, elementary refactorings that are used to compose others. However, they do not mention how the composition works in practice and what is the set of microrefactorings. In the work presented in this article, we describe all the transformations, refactorings, and elementary operations.

Reichenbach *et al.,* [18] propose to use postconditions on a model of the code to check whether refactorings are behavior-preserving. They also introduce the notion of program metamorphosis steps as elementary non-behavior preserving units that can be composed to define refactorings. In their goal, program metamorphosis steps look similar to transformations, however, their implementation or design is not clearly described. We note that they do not have applicability preconditions. They can manipulate ill-formed code, and one of them even allows pasting arbitrary code from the clipboard, something that transformations do not allow.

Kniesel *et al.,* [15] focus on the composition of refactorings. Their conditional transformations are more generic than the transformations presented in this article, however, they do have preconditions too. The authors introduce And (*e.g.,* sequence of refactorings) and Or. Based on this, they compute the validation of composed refactoring preconditions before their execution. They propose a formal model for automatic, program-independent composition of conditional program transformations. They show that conditional transformations, including refactorings, can be composed of a limited set of basic operations. Program-independent derivation of a precondition for the composite is based on the notion of "transformation description", which can be seen as a simplified, yet equally powerful, variant of Roberts' "postconditions". It should be noted that while Roberts uses postconditions in his thesis, the implementation never contained post conditions. In addition, we did not start to work on refactoring composition even if we started to see some composition patterns during the implementation of the architecture presented in this paper.

Ò Cinnéide *et al.,* [45, 46] propose a methodology for developing design pattern transformations and a prototype tool that automatically performs them. The authors define custom transformations that introduce design patterns in existing code bases. They rely on low-level refactorings to develop design pattern transformations. This paper mostly discusses the composition of smaller refactorings and custom transformations to create design pattern transformations, and behavior preservation in that context, while our work focuses on the lower-level details and reuse in the underlying refactoring engine implementation.

Independent and cross languages.. While the definition of language-independent or cross-language refactorings does not focus on the reuse of transformation logic, they are the only work besides the Ph.D. of D. Roberts formalizing refactoring implementation. Tichelaar [47, 48] presents some language-independent refactorings on top of the FAMIX metamodel [49] while Mayer *et al.,* present a metamodel to support cross-language refactorings [50]. Such approaches are interesting because they focus on the implementation of the refactorings. Nevertheless, they do not provide an analysis of the reuse of transformation and composition of refactorings.

Horpácsi *et al.,* [19] propose a framework to define trustworthy refactorings that can be parametrized by languages. They propose schemes that are language refactoring idioms aka transformation templates which are parametrized by conditional term rewrite rules. They support refactoring compositionality using basic imperative controls such as sequencing, branching, and iteration.

Butler *et al.,* [51] propose a concept of cascaded refactoring to be applied when refactoring frameworks.

Refactoring engines.. There is some work on refactoring engines for languages such as Erlang with Tidier [52, 53], Wrangler [54] and RefactorErl [55].

Li *et al.,* [56] present two approaches to developing a refactoring engine taken by two teams. One of the teams developed the Wrangler engine by using an annotated abstract syntax tree. The other team developed what is later known as RefactorErl by using a relational database to store both abstract syntax trees and semantic information. The paper discusses the representation of the program, while our work is focused on the implementation of refactorings, preconditions, and user interaction. Note that in section 3.1 we analyzed and discussed the representation that is the basis of our work since we extended and refactored the existing refactoring engine.

Horváth *et al.,* [57] present RefactorErl, a refactoring tool for the Erlang programming language. The authors describe a major redesign of the tool and provide important insights for developing refactoring tools. RefactorErl works on a graph representation of the program, therefore most of the insights are specific to graph-based refactoring engines and are not translatable to non-graph-based refactoring engines.

Horpácsi *et al.,* [58] defined a DSL that enables descriptive, higher-level definitions of refactorings that are also executable. The authors differentiate between prime refactorings (ones that cannot be decomposed into smaller refactorings) and composite refactorings. This work is focused more on describing refactorings, whereas our work is focused on the design and architecture of a complete refactoring engine. Their concepts of prime and composite refactorings may seem similar to our transformations and refactorings. However, there is a key difference - our refactorings preserve behavior, and transformations do not. The authors also covered behavior preservation through the verification of refactoring. Using reachability logic, they verified refactoring definitions. However, that proof system is not complete, and because of that the authors presented an additional method for proving the correctness of a single application of the refactoring.

RubyMine from JetBrains[5] offers a limited amount of refactorings (Rename, MoveAndCopy, Extract Method, Extract Field, Extract superclass, Extract Parameter Inline, Pull members up and down, Safe Delete). To the best of our knowledge, there is, however, no explanation or information about the actual implementation of the refactoring engine. This engine is driven by the user interface. Refactorings are simply explained from a user perspective.

Kim *et al.,* [26] proposed a new architecture for a refactoring engine called R3 for Java. They wanted to address the limits of the Java refactoring engine (slow refactoring performance). Their proposed architecture is similar to that of the Refactoring browser in Pharo, which we studied in this paper. It is similar in the sense that R3 uses an in-memory model of the program and is not exclusively manipulating ASTs. The R3 model is a kind of direct database schema with foreign keys. In addition, it contains information about the program entities encoded as boolean.

Borba *et al.,* [59] discuss refactorings in the context of aspect-oriented programming. They do make a distinction between refactorings and code transformations, where refactorings preserve behavior without adding new features. However, nothing is said about code reuse between these two.

There is also a large body of research on C refactorings where one of the important challenges is handling the preprocessor [60, 61, 62]. Garrido *et al.,* defined one of the first C refactoring engines that correctly deals with preprocessor commands [63]. CScout [64] is a refactoring engine for C that correctly handles the preprocessor for a large number of independent program families. It can detect dead objects to remove and automatically perform four refactorings that it supports: rename identifier, add parameter, remove parameter, and change parameter order.

Code to code transformation.. Some work such as in [15, 65] focuses on the derivation of a composite refactoring precondition from its constituents. The idea is to execute the composite preconditions before performing the actual code transformation.

Li and Thomson [22, 66] present a domain-specific language (DSL) that extends the existing refactoring engine, Wrangler. The DSL can be used to define new refactorings in Erlang for Erlang. This is one of the rare articles discussing the notion of atomic and non-atomic (composite) refactorings. During composition, and as a design choice, Wrangler does not do anything to derive a composite refactoring precondition. Instead, each primitive refactoring is executed individually. With Wrangler, a primitive refactoring is extended with a refactoring command generator. Then they introduced a DSL that enables users to have fine control over the generation of refactoring commands and the interaction between the user and the refactoring engine [22]. One can draw a parallel between this DSL and the drivers presented in this paper. The difference is that DSL scripts represent the functional style of programming where

[5]https://www.jetbrains.com/help/ruby/refactoring-source-code.html

they contain all the information of a custom or composite refactoring (user interaction, conditions for application, and how to perform refactorings), whereas with the driver we leverage OOP and delegate refactoring precondition checking and implementation to the refactorings themselves, while the driver takes care of user interaction and invoking appropriate refactorings.

Hills *et al.* [24] show how the authors integrate Eclipse and Rascal to perform a transformation from a visitor to an interpreter design pattern. They use Rascal as a meta-programming environment.

Clang offers a kind of minimalistic source code manipulation engine[6]. It can be used to define source change transformations using an AST matcher. It offers two kinds of actions: *source changes* and *symbol* occurrences. The rule SourceChangeRefactoringRule produces source replacements that are applied to the source files. Usually, the changes are located in a single 'translation' unit. FindSymbolOccurrencesRefactoringRule produces a set of occurrences that refer to a particular symbol. The documentation mentions that FindSymbolOccurrencesRefactoringRule can be *'used to implement an interactive renaming action that allows users to specify which occurrences should be renamed during the refactoring'*. However, it is unclear how symbols are mapped to a model of the program. This is probably left to the user to manage all the modeling logic to distinguish between classes, fields, and methods as well as representing hierarchies. It is unclear if there is support for preconditions. To the best of our knowledge, this is left to the user and this engine is a source code transformation API more than a refactoring one.

User and usability.. Boshernotan *et al.* [67, 29] propose a program manipulation paradigm that enables programmers to change source code with interactively-constructed visual program transformations. Similarly, Rizun *et al.* [68] propose direct manipulation of AST nodes to generate corresponding code transformations using Refactoring Parse Tree Rewriter [4]. Vakilian *et al.,* [10] propose a new paradigm for refactorings based on the composition of small refactorings. They argue that a compositional paradigm where users perform small automated refactorings that they invoke manually, instead of one big automated refactoring leads to better user feedback and overall more use of refactorings.

Semantics-driven.. Kesseli, in his PhD [69], explores semantics-driven refactorings in opposition to syntactic refactorings (the ones considered in this paper). He presents and implements a program synthesis algorithm based on the CEGIS paradigm and demonstrates that it can be applied to a diverse set of applications. It does not discuss, however, the reuse of refactoring logic.

Refactoring detection and mining.. Some publications focus on identifying the application of refactoring (Extract method application [70]), general refactorings [12]) with tools such as RefactoringMiner2.0. Other publications mine missed

[6]https://clang.llvm.org/docs/RefactoringEngine.html

opportunities to refactor code (move method [13], missed polymorphism [14].) The work presented in this article is concerned with the implementation and in particular the reuse of logic between transformations and refactorings — not the applications of refactorings on an existing code base.

Bibiano *et al.,* [71] mine composite refactorings in software repositories: they mine the application of a sequence of refactorings touching the same source code element. They focused on smell removal. Brito *et al.,* [72] extend this work.

Such studies do not analyze the refactoring semantics nor the reuse of refactorings or transformation logic to define larger ones, but they study empirically how developers use multiple refactorings on the same source element. The term "composite" in their work is different from the actual composition of refactorings by the refactoring engine as discussed in [31].

Abdullah AlOmar [73] researched how developers document refactoring activities. The author presents a model that detects inconsistencies between commit messages and developer-related refactoring events, as well as a procedure for documenting refactorings.

Bavota *et al.,* [74] mined three Java systems and investigated when refactorings introduce faults.

Refactoring opportunities automatic identification.. Foster *et al.,* [75] created a system called *WitchDoctor* that detects when a user is performing a refactoring by hand and then offers to complete it. Similarly, Ge *et al.,* [11] created *BeneFactor*, a tool to recognize when a developer is doing manual refactoring and offers to automatically complete it.

Ò Cinnéide *et al.,* [76] propose a tool that automatically performs refactorings on a code base, and they evaluated it through a survey to check whether it reduced the difficulty of writing tests.

Kataoka *et al.,* [77] propose Daikon, a program invariant detector. When a certain invariant holds a specific refactoring is applicable.

8. Conclusion

The extensive analysis of an existing large library of refactorings and transformations done by Anquetil *et al.,* [31] was a call for a better refactoring architecture. This article presents a new architecture for the refactoring engine, inherited from the implementation of D. Roberts and J. Brant [2, 3, 4].

This new architecture supports two important scenarios: interactive use and scripting, (i.e., batch use). It does this by introducing Drivers, which are objects that provide guidance to developers during the application of refactorings. In interactive mode, this new architecture reduces logic duplication by introducing partial initialization of refactorings. Additionally, it defines a clear API that unifies refactorings and transformations, and expresses refactorings as decorators over transformations. It also formalizes the use of different kinds of preconditions, thus providing better user feedback.

Current results show that elementary transformations such as the transformation Add Method are reused in 24 refactorings (including the refactoring Add Method) and 11 other transformations. The transformation Remove Method is reused in 11 refactorings (including the refactoring Remove Method) and 7 other transformations.

Our future work is to migrate all the existing refactorings and transformations to the new architecture. Then, we will focus on supporting application developers in defining their own, often domain-specific, transformations/refactorings by proposing a library of composition operators.

Acknowledgments.. This work was partially supported by COST Action CA19135 CERCIRAS: Connecting Education and Research Communities for an Innovative Resource Aware Society funded by the European Union through the COST Association. We want to thank the anonymous reviewers of the early version of the article as well as the constant support from the Pharo industrial consortium http://consortium.pharo.org. This research has been supported by the Ministry of Science, Technological Development and Innovation (Contract No. 451-03-65/2024-03/200156) and the Faculty of Technical Sciences, University of Novi Sad through project "Scientific and Artistic Research Work of Researchers in Teaching and Associate Positions at the Faculty of Technical Sciences, University of Novi Sad" (No. 01-3394/1).

References

[1] W. F. Opdyke, Refactoring object-oriented frameworks, Ph.D. thesis, University of Illinois (1992).

[2] D. Roberts, J. Brant, R. E. Johnson, B. Opdyke, An automated refactoring tool, in: Proceedings of ICAST '96, 1996.

[3] D. Roberts, J. Brant, R. E. Johnson, A refactoring tool for Smalltalk, Theory and Practice of Object Systems (TAPOS) 3 (4) (1997) 253–263.

[4] J. Brant, D. Roberts, "Good Enough" Analysis for Refactoring, in: Object-Oriented Technology Ecoop '98 Workshop Reader, LNCS, Springer-Verlag, 1998, pp. 81–82.

[5] D. B. Roberts, Practical analysis for refactoring, Ph.D. thesis, University of Illinois (1999).

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: Improving the Design of Existing Code, Addison Wesley, 1999.

[7] E. Murphy-Hill, C. Parnin, A. P. Black, How we refactor, and how we know it, IEEE Transactions on Software Engineering 38 (1) (2011) 5–18.

[8] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, D. Dig, A comparative study of manual and automated refactorings, in: 27th European Conference on Object-Oriented Programming, 2013, pp. 552–576.

[9] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, R. E. Johnson, Use, disuse, and misuse of automated refactorings, in: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 233–243.
URL http://dl.acm.org/citation.cfm?id=2337223.2337251

[10] M. Vakilian, N. Chen, R. Z. Moghaddam, S. Negara, R. E. Johnson, A compositional paradigm of automating refactorings, in: European Conference on Object-Oriented Programming, 2013, pp. 527–551.

[11] X. Ge, Q. L. DuBose, E. Murphy-Hill, Reconciling manual and automatic refactoring, in: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 211–221.
URL http://dl.acm.org/citation.cfm?id=2337223.2337249

[12] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, D. Dig, Accurate and efficient refactoring detection in commit history, in: Proceedings of the 40th International Conference on Software Engineering (ICSE '18), ACM, New York, NY, USA, 2018, pp. 483–494. doi:10.1145/3180155.3180206.

[13] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Transactions on Software Engineering 35 (3) (2009) 347–367.

[14] N. Tsantalis, A. Chatzigeorgiou, Identification of refactoring opportunities introducing polymorphism, Journal of Systems and Software 83 (3) (2010) 391–404.

[15] G. Kniesel-Wünsche, H. Koch, Static composition of refactorings, Science of Computer Programming 52 (2004) 9–51. doi:10.1016/j.scico.2004.03.002.

[16] T. Schäfer, J. Jonas, M. Mezini, Mining framework usage changes from instantiation code, in: International Conference on Software Engineering (ICSE), ACM, New York, NY, USA, 2008, pp. 471–480. doi:10.1145/1368088.1368153.

[17] M. Schäfer, M. Verbaere, T. Ekman, O. de Moor, Stepping stones over the refactoring rubicon – lightweight language extensions to easily realise refactorings, in: S. Drossopoulou (Ed.), European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag, 2009, pp. 369–393.

[18] C. Reichenbach, D. Coughlin, A. Diwan, Program metamorphosis, in: S. Drossopoulou (Ed.), European conference on Object-Oriented Programming, Springer Berlin Heidelberg, 2009, pp. 394–418.

[19] D. Horpácsi, J. Köszegi, D. J. Németh, Towards a generic framework for trustworthy program refactoring, Acta Cybernetica 25 (4) (2021) 753–779.

doi:10.14232/actacyb.284349.
URL https://cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/4121

[20] P. Tesone, G. Polito, L. Fabresse, N. Bouraqadi, S. Ducasse, Dynamic software update from development to production, Journal of Object Technology 17 (2018) 1–36. doi:10.5381/jot.2018.17.1.a2.

[21] M. Verbaere, R. Ettinger, O. de Moor, Jungl: a scripting language for refactoring, in: Proceedings of International Conference on Software Engineering, 2006.

[22] H. Li, S. Thompson, A domain-specific language for scripting refactorings in erlang, in: FASE, 2012.

[23] F. Steimann, J. von Pilgrim, Constraint-based refactoring with foresight, in: ECOOP, 2012.

[24] M. Hills, P. Klint, J. J. Vinju, Scripting a refactoring with rascal and eclipse, in: 5th Workshop on Refactoring Tools, 2012, pp. 40–49.

[25] J. Kim, D. Batory, D. Dig, Scripting parametric refactorings in java to retrofit design patterns, in: ICSME, 2015.

[26] J. Kim, D. Batory, D. Dig, M. Azanza, Improving refactoring speed by 10x, in: Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 1145 – 1156. doi:10.1145/2884781.2884802.

[27] G. Florijn, M. Meijers, P. van Winsen, Tool support for object-oriented patterns, in: M. Aksit, S. Matsuoka (Eds.), Proceedings ECOOP '97, Vol. 1241 of LNCS, Springer-Verlag, Jyvaskyla, Finland, 1997, pp. 472–495.

[28] G. Santos, N. Anquetil, A. Etien, S. Ducasse, M. T. Valente, System specific, source code transformations, in: 31st IEEE International Conference on Software Maintenance and Evolution, 2015, pp. 221–230.

[29] M. Boshernitsan, L. S. Graham, A. M. Hearst, Aligning development tools with the way programmers think about code changes, in: Conference on Human Factors in Computing Systems (CHI '07), 2007. doi:10.1145/1240624.1240715.

[30] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, M. Denker, Pharo by Example, Square Bracket Associates, Kehrsatz, Switzerland, 2009.
URL http://books.pharo.org

[31] N. Anquetil, M. Campero, S. Ducasse, J.-P. Sandoval, P. Tesone, Transformation-based refactorings: a first analysis, in: International Workshop of Smalltalk Technologies, 2022.

[32] I. Thomas, S. Ducasse, P. Tesone, G. Polito, Pharo: a reflective language - A first systematic analysis of reflective APIs, in: IWST 23 - International Workshop on Smalltalk Technologies, Lyon, France, 2023.
URL https://inria.hal.science/hal-04217271

[33] V. Uquillas Gómez, S. Ducasse, T. D'Hondt, Ring: a unifying meta-model and infrastructure for Smalltalk source code analysis tools, Journal of Computer Languages, Systems and Structures 38 (1) (2012) 44–60.

[34] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[35] A. Ouni, M. Kessentini, M. Ò Cinnéide, H. Sahraoui, K. Deb, K. Inoue, More: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells, Journal of Software: Evolution and Process 29 (Mar. 2017). doi:10.1002/smr.1843.

[36] A. Etien, A. Muller, T. Legrand, X. Blanc, Combining independent model transformations, in: Proceedings of the 2010 ACM Symposium on Applied Computing, 2010, pp. 2237–2243.

[37] A. Etien, A. Muller, T. Legrand, R. F. Paige, Localized model transformations for building large-scale transformations, Software & Systems Modeling 14 (3) (2015) 1189–1213. doi:10.1007/s10270-013-0379-8.

[38] T. Mens, S. Demeyer, B. Du Bois, H. Stenten, P. Van Gorp, Refactoring: Current research and future trends, in: LDTA@ETAPS, 2003.
URL https://api.semanticscholar.org/CorpusID:9433418

[39] T. Mens, T. Tourwé, A survey of software refactoring, IEEE Transaction on Software Engineering 30 (2) (2004) 126–139. doi:10.1109/TSE.2004.1265817.

[40] S. Akhtar, M. Nazir, A. Ali, A. Khan, M. Atif, A systematic literature review on software - refactoring techniques, challenges, and practices (Mar. 2022). doi:10.21203/rs.3.rs-1472519/v1.

[41] A. A. B. Baqais, M. Alshayeb, Automatic software refactoring: a systematic literature review, Software Quality Journal 28 (2) (2020) 459–502. doi:10.1007/s11219-019-09477-y.

[42] M. Schäfer, O. de Moor, Specifying and implementing refactorings, in: Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'10), 2010.

[43] M. Schäfer, T. Ekman, O. de Moor, Sound and extensible renaming for java, in: G. Kiczales (Ed.), Object-Oriented Programming, Systems and Languages (OOPSLA), ACM Press, 2008, pp. 227–294.

[44] M. Schäfer, J. Dolby, M. Sridharan, F. Tip, E. Torlak, Correct refactoring of concurrent java code, in: T. D'Hondt (Ed.), European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag, 2010.

[45] M. Ò Cinnéide, P. Nixon, A methodology for the automated introduction of design patterns, in: Proceedings of the IEEE International Conference on Software Maintenance, ICSM '99, IEEE Computer Society, USA, 1999, p. 463.

[46] M. Ò Cinnéide, Composite refactorings for java programs, in: Proceedings of the Workshop on Formal Techniques for Java Programs, European Conference on Object-Oriented Programming, 2000.

[47] S. Tichelaar, S. Ducasse, S. Demeyer, O. Nierstrasz, A meta-model for language-independent refactoring, in: Proceedings of International Symposium on Principles of Software Evolution, ISPSE'00, IEEE Computer Society Press, 2000, pp. 157–167. doi:10.1109/ISPSE.2000.913233.

[48] S. Tichelaar, Modeling object-oriented software for reverse engineering and refactoring, Ph.D. thesis, University of Bern (Dec. 2001).
URL http://scg.unibe.ch/archive/phd/tichelaar-phd.pdf

[49] S. Ducasse, N. Anquetil, U. Bhatti, A. Cavalcante Hora, J. Laval, T. Girba, MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family, Tech. rep., RMod – INRIA Lille-Nord Europe (2011).

[50] P. Mayer, A. Schroeder, W. Löwe, Cross-language code analysis and refactoring, in: In Proceedings of the International Workshop on Source Code Analysis and Manipulation, 2012. doi:10.1109/SCAM.2012.11.

[51] G. Butler, L. Xu, Cascaded refactoring for framework, in: ACM SIGSOFT Symposium on Software Reusability, 2001.
URL https://api.semanticscholar.org/CorpusID:14330253

[52] K. Sagonas, T. Avgerinos, Automatic refactoring of erlang programs, in: A. Porto, F. J. López-Fraguas (Eds.), International Conference on Principles and Practice of Declarative Programming, ACM, 2009, pp. 13–24. doi:10.1145/1599410.1599414.
URL https://doi.org/10.1145/1599410.1599414

[53] T. Avgerinos, K. Sagonas, Cleaning up erlang code is a dirty job but somebody's gotta do it, in: C. B. Earle, S. J. Thompson (Eds.), 8th Workshop on Erlang, ACM, 2009, pp. 1–10. doi:10.1145/1596600.1596602.
URL https://doi.org/10.1145/1596600.1596602

[54] H. Li, S. Thompson, G. Orosz, M. Tóth, Refactoring with wrangler, updated: Data and process refactorings, and integration with eclipse, in: Workshop on ERLANG, Association for Computing Machinery, New York, NY, USA, 2008, pp. 61–72. doi:10.1145/1411273.1411283.
URL https://doi.org/10.1145/1411273.1411283

[55] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Koszegi, M. Tejfel, M. Tóth, Refactorerl - source code analysis and refactoring in erlang, in: Proceedings of the 12th Symposium on Programming Languages and Software Tools, ISBN 978-9949-23-178-2, Tallin, Estonia, 2011, pp. 138–148.

[56] H. Li, S. Thompson, L. Lövei, Z. Horváth, T. Kozsik, A. Víg, T. Nagy, Refactoring erlang programs, in: The Proceedings of 12th International Erlang/OTP User Conference, 2006.

[57] Z. Horváth, L. Lövei, T. Kozsik, R. Kitlei, A. N. Víg, T. Nagy, M. Tóth, R. Király, Building a refactoring tool for erlang, in: Workshop on Advanced Software Development Tools and Techniques, WASDETT, Vol. 2008, 2008.

[58] D. Horpácsi, J. Kőszegi, S. Thompson, Towards trustworthy refactoring in erlang, arXiv preprint arXiv:1607.02228 (2016).

[59] P. Borba, S. Soares, Refactoring and code generation tools for aspectj, in: Proc. of the Workshop on Tools for Aspect-Oriented Software Development (with OOPSLA), 2002.

[60] A. Garrido, R. Johnson, Challenges of refactoring c programs, in: Proceedings of the international workshop on Principles of software evolution, ACM, 2002, pp. 6–14.

[61] D. Spinellis, Global analysis and transformations in preprocessed languages, IEEE Transactions on Software Engineering 29 (11) (2003) 1019–1030.

[62] A. Garrido, R. Johnson, Analyzing multiple configurations of a c program, in: Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on, IEEE, 2005, pp. 379–388.

[63] A. Garrido, R. Johnson, Embracing the c preprocessor during refactoring, Journal of Software: Evolution and Process 25 (12) (2013) 1285–1304.

[64] D. Spinellis, Cscout: A refactoring browser for c, Science of Computer Programming 75 (4) (2010) 216–231.

[65] M. Ò Cinnéide, Automated application of design patterns : a refactoring approach, Ph.D. thesis, Trinity College - School of Computer Science and Statistics (2001).

[66] H. Li, S. Thompson, Let's make refactoring tools user-extensible!, in: Proceedings of the fifth workshop on refactoring tools, 2012, pp. 32–39.

[67] M. Boshernitsan, S. L. Graham, ixj: Interactive source-to-source transformations for java, in: OOPSLA Companion, 2004.

[68] M. Rizun, J.-C. Bach, S. Ducasse, Code transformation by direct transformation of asts, in: International Workshop on Smalltalk Technologies (IWST), 2015.

[69] P. Kesseli, Semantic refactorings, Ph.D. thesis, University of Oxford (2018).

[70] M. Fokaefs, N. Tsantalis, E. Stroulia, A. Chatzigeorgiou, Identification and application of Extract Class refactorings in object-oriented systems, Journal of Systems and Software 85 (10) (2012) 2241–2260. doi:10.1016/j.jss.2012.04.013.
URL https://linkinghub.elsevier.com/retrieve/pii/S0164121212001057

[71] A. C. Bibiano, W. K. G. Assuncao, D. Coutinho, K. Santos, V. Soares, R. Gheyi, A. Garcia, B. Fonseca, M. Ribeiro, D. Oliveira, C. Barbosa, J. L. Marques, A. Oliveira, Look ahead! revealing complete composite refactorings and their smelliness effects, in: International Conference on Software Maintenance and Evolution (ICSME), 2021, pp. 298–308. doi:10.1109/ICSME52107.2021.00033.

[72] A. Brito, A. Hora, M. Tulio Valente, Towards a catalog of composite refactorings, Journal of Software: Evolution and Process (2023). doi:https://doi.org/10.1002/smr.2530.

[73] E. Abdullah AlOmar, State of refactoring adoption: Better understanding developer perception of refactoring, in: 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), IEEE, Melbourne, Australia, 2023, pp. 635–639. doi:10.1109/MSR59073.2023.00090.
URL https://ieeexplore.ieee.org/document/10174163/

[74] G. Bavota, A. Qusef, R. Oliveto, A. D. Lucia, D. Binkley, An empirical analysis of the distribution of unit test smells and their impact on software maintenance, in: International Conference on Software Maintenance (ICSM), IEEE, 2012, pp. 56–65. doi:10.1109/ICSM.2012.6405253.

[75] S. R. Foster, W. G. Griswold, S. Lerner, Witchdoctor: Ide support for real-time auto-completion of refactorings, in: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 222–232.
URL http://dl.acm.org/citation.cfm?id=2337223.2337250

[76] M. Ò Cinnéide, D. Boyle, I. H. Moghadam, Automated refactoring for testability, in: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, 2011, pp. 437–443. doi:10.1109/ICSTW.2011.23.

[77] Y. Kataoka, M. D. Ernst, W. G. Griswold, D. Notkin, Automated support for program refactoring using invariants, in: Proceedings of the International Conference on Software Maintenance, (Florence, Italy), 2001, pp. 736–743.

Appendix A. Original list of refactorings

Original list of refactorings as in [5] in alphabetical order:

- Abstract Class Variable
- Abstract Instance Variable
- Add Class
- Add Class Variable
- Add Instance Variable
- Add Parameter to Method
- Convert Superclass to Sibling
- Convert Temporary to Instance Variable
- Create Accessors for Class Variable
- Create Accessors for Instance Variable
- Extract Code as Method
- Extract Code as Temporary
- Inline Call
- Inline Temporary
- Move Method to Component
- Move Temporary to Inner Scope
- Protect Instance Variable
- Push Up/Down Class Variable
- Push Up/Down Instance Variable
- Push Up/Down Method
- Remove Class
- Remove Class Variable
- Remove Instance Variable
- Remove Method
- Remove Parameter from Method
- Rename Class
- Rename Class Variable
- Rename Instance Variable
- RenameMethod
- Rename Temporary

Appendix B. Refactorings added in Legacy Pharo (prior to our work, Pharo10)

- Abstract Variables
- Accessor Class
- AddMethod
- Category Regex
- Class Regex
- Copy Class
- Copy Package
- Create Accessors With Lazy Initialization For Variable
- Create Cascade
- Deprecate Class
- Deprecate Method
- Expand Referenced Pools
- Extract Method And Occurrences
- Extract Method To Component
- Extract SetUp Method And Occurrences
- Extract SetUp Method
- Find And Replace
- Find And Replace SetUp
- Generate EqualHash
- Generate PrintString
- Inline AllSenders
- Inline Method From Component
- Inline Parameter
- Merge Instance Variable Into Another
- Move Inst Var To Class
- Move Method To Class
- Move Method To Class Side
- Move Variable Definition
- Protect Instance Variable
- Protocol Regex
- Realize Class
- Remove All Senders
- Remove Class Keeping Subclasses
- Remove HierarchyMethod
- Remove Sender
- Rename Package
- Replace Method
- Source Regex
- Split Cascade
- Split Class
- Swap Method

Appendix C. Transformations added in Legacy Pharo (prior to our work, Pharo10)

- Abstract Variables
- Add Accessors For Class
- Add Assignment
- Add Class
- Add Class Comment
- Add Message Send
- Add Method
- Add Method Comment
- Add Parameter
- Add Pragma
- Add Protocol
- Add Return Statement
- Add Subtree
- Add Temporary Variable
- Add Variable
- Add Variable Accessor
- Add Variable Accessor With Lazy Initialization
- Change Method Name
- Deprecate Class
- Deprecate Method
- Expand Referenced Pools
- Extract Method
- Extract To Temporary
- Inline Method
- Inline Temporary
- Merge Instance Variable Into Another
- Method Protocol
- Move Class
- Move Instance Variable To Class
- Move Method
- Move Method To Class
- Move Method To Class Side
- Move Temporary Variable Definition
- Protect Variable
- Pull Up Method
- Pull Up Variable
- Push Down Method
- Push Down Variable
- Realize Class
- Remove Assignment
- Remove Class
- Remove Direct Access To Variable
- Remove Hierarchy Method
- Remove Message Send
- Remove Method
- Remove Parameter
- Remove Pragma
- Remove Protocol
- Remove Return Statement
- Remove Subtree
- Remove Temporary Variable
- Remove Variable
- Rename And Deprecate Class
- Rename Class
- Rename Method
- Rename Package
- Rename Temporary Variable
- Rename Variable
- Replace Subtree
- Split Class
- Temporary To Instance Variable

Appendix D. Analysis of refactorings and their preconditions

Refactoring name	**Directly defines applicability preconditions**	**Has indirect applicability preconditions**	**Directly defines behavior-preserving preconditions**	**Has indirect behavior-preserving preconditions**
Abstract Class Variable References	✗	✗	✗	✗
Abstract Instance Variable References	✓	✓	✗	✗
Abstract Variables	✗	✗	✓	✓
Add Class Variable	✓	✓	✓	✓
Add Instance Variable	✓	✓	✓	✓
Add Method	✓	✓	✓	✓

Continued on next page

Refactoring name	**Directly defines applicability preconditions**	**Has indirect applicability preconditions**	**Directly defines behavior-preserving preconditions**	**Has indirect behavior-preserving preconditions**
Add Parameter	✓	✓	✗	✓
Children To Siblings	✓	✓	✗	✓
Copy Class	✓	✓	✗	✓
Copy Package	✓	✓	✗	✓
Create Cascade	✓	✓	✓	✓
Deprecate Class	✓	✓	✗	✗
Expand Referenced Pools	✗	✗	✓	✓
Extract Method And Occurrences	✗	✓	✓	✓
Extract Method To Component	✗	✓	✗	✓
Extract Method	✓	✓	✓	✓
Extract SetUp Method And Occurrences	✗	✓	✓	✓
Extract SetUp Method	✓	✓	✓	✓
Extract To Temporary Variable	✓	✓	✗	✗
Inline All Senders	✓	✓	✗	✓
Inline Method From Component	✓	✓	✓	✓
Inline Method	✓	✓	✓	✓
Inline Parameter	✓	✓	✓	✓
Inline Temporary	✓	✓	✗	✗
Insert New Class	✓	✓	✗	✗
Merge Instance Variable Into Another	✓	✓	✗	✗
Move Method To Class	✓	✓	✗	✗
Move Method To Class Side	✓	✓	✓	✓
Move Method	✓	✓	✓	✓
Move Variable Definition	✓	✓	✗	✗
Protect Instance Variable	✓	✓	✗	✓
Pull-Up Class Variable	✓	✓	✓	✓
Pull-Up Instance Variable	✓	✓	✓	✓
Pull-Up Method	✓	✓	✓	✓
Push Down Class Variable	✓	✓	✓	✓
Push Down Instance Variable	✓	✓	✓	✓
Push Down Method	✓	✓	✓	✓
Remove All Senders	✗	✓	✓	✓
Remove Class And Reparent Subclasses	✓	✓	✓	✓
Remove Class Pushing State To Subclasses	✗	✓	✓	✗
Remove Methods in Hierarchy	✓	✓	✗	✓
Remove Method	✓	✓	✓	✓
Remove Parameter	✓	✓	✗	✓
Remove Instance Variable	✓	✓	✓	✓
Remove Methods	✓	✓	✓	✓
Remove Shared Variable	✓	✓	✓	✓
Remove Sender	✓	✓	✓	✓
Rename Argument Or Temporary	✓	✓	✗	✗
Rename Instance Variable	✓	✓	✗	✗
Rename Class	✓	✓	✗	✗
Rename Method	✓	✓	✓	✓

Continued on next page

Refactoring name	**Directly defines applicability preconditions**	**Has indirect applicability preconditions**	**Directly defines behavior-preserving preconditions**	**Has indirect behavior-preserving preconditions**
Rename Package	✓	✓	✗	✗
Rename Shared Variable	✓	✓	✗	✗
Split Cascade Message	✓	✓	✗	✗
Split Class	✓	✓	✗	✓
Temporary To Instance Variable	✓	✓	✓	✗

Table D.2: Analysis of preconditions in refactorings. Column one displays the refactoring name, column two shows a check mark if a refactoring has directly defining applicability preconditions, and column three shows a check mark if a refactoring has indirect applicability preconditions (through composition). Columns four and five show direct and indirect behavior-preserving preconditions respectively.