# Transformation-based Refactorings: a First Analysis

N. Anquetil[1], M. Campero[1], Stéphane Ducasse[1], J.-P. Sandoval Alcocer[2] and P. Tesone[1]

[1]Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 - CRIStAL, F-59000 Lille, France
[2]Pontificia Universidad Catolica de Chile, Santiago, Chile

**Abstract**

Refactorings are behavior preserving transformations. Little work exists on the analysis of their *implementation* and in particular how refactorings could be composed from smaller, reusable, parts (being simple transformations or other refactorings) and how (non behavior preserving) transformations could be used in isolation or to compose new refactoring operators. In this article we study the seminal implementation and evolution of Refactorings as proposed in the PhD of D. Roberts. Such an implementation is available as the Refactoring Browser package in Pharo. In particular we focus on the possibilities to reuse transformations independently from the behavior preserving aspect of a refactoring. The long term question we want to answer is: Is it possible to have more atomic transformations and refactorings composed out of such transformations? We study pre-conditions of existing refactorings and identify several families. We identify missed opportunities of reuse in the case of implicit composite refactorings. We analyze the refactorings that are explicitly composed out of other refactorings to understand whether the composition could be expressed at another level of abstraction. This analysis should be the basis for a more systematic expression of composable refactorings as well as the reuse of logic between transformations and refactorings.

## 1. Introduction

Refactorings are behavior preserving code transformations. The seminal work of Opdyke [Opd92] and the Refactorings Browser (first implementation of Refactorings of Roberts and Brant [RBJO96, RBJ97, BR98]) paved the way to the spread of refactorings [FBB+99]. They are now a must-have standard in modern IDEs [MHPB11, NCV+13, VCN+12, VCM+13, GDMH12]. A lot of research has been performed on refactorings such as for their detection [TME+18], missed application opportunities [TC09, TC10], practitioner use [MHPB11, VCN+12, NCV+13, VCM+13], or atomic refactorings for live environments [TPF+18]. Several publications focus on scripting refactorings [VEdM06, LT12, SvP12, HKV12, KBD15]. Finally, some work tried to speed up refactoring engines, proposing alternatives to the slow and bogus Java refactoring engine [KBDA16]. Related to this, it should be noted that the Pharo Refactoring Browser architecture supports fast pre-condition validation and refactoring execution and does not suffer from the architecture problems reported by Kim et al. [KBDA16].

*IWST'22: International Conference of Smalltalk Technologies, August 24–26, 2022, Novy Sad, Serbia*
nicolas.anquetil@inria.fr (N. Anquetil); stephane.ducasse@inria.fr (S. Ducasse); juanpablo.sandoval@ing.puc.cl (J.-P. S. Alcocer); pablo.tesone@inria.fr (P. Tesone)
0000-0002-5615-6691 (N. Anquetil); 0000-0001-6070-6599 (S. Ducasse); 0000-0002-8335-4351 (J.-P. S. Alcocer); 0000-0002-5615-6691 (P. Tesone)



CEUR Workshop Proceedings (CEUR-WS.org)

Still, from a daily development perspective, refactorings and their behavior preserving form are not enough [SAE+15]. Non-behavior preserving code transformations are also needed [BGH07]. For example, consider replacing all the invocations of a given message by another one (that we might name Replace Call(msg1,msg2)). It should update all the msg1 invocations to msg2 invocations. Such transformation might not preserve behavior, yet it is a need that arises in real development situations. It is clear that Replace Call has strong similarities with the Rename Method refactoring, but it would be awkward - if not impossible depending on the refactoring engine, for a developer to perform it by applying Rename Method. When in need of such transformation of the source code, a developer is left to perform the changes manually or with a code rewriting engine that can be cumbersome to use [SAE+15].

Defining some specific code transformations such as Replace Call and letting the developer define its own transformation are our long term engineering goal. While targeting this goal, we wish to shed a new light on the following related questions:

- Could both refactorings and transformations share their code modification logic?
- Could we decouple code transformations from refactorings to be able to reuse them when behavior preservation is not a concern?
- Could we compose refactorings (existing or new) from such code transformations?
- Could complex refactorings be expressed out of simpler ones?

This article is focusing on the duality of refactorings (with their pre-conditions) and transformations, and how they can be composed together [BGH07]. Although it may seem trivial to state that refactorings can be decomposed in pre-conditions + transformations, the results of this preliminary study are that (i) some pre-conditions will always be needed, even for non-behavior preserving transformations, and (ii) there are implementation issues in current Pharo Refactoring Browser (*e.g.,* part of the transformation logic implemented in the pre-condition) that would prevent this decomposition and thus the use of transformations alone.

Our contributions are the following:

- An analysis of the original implementation of Refactorings. We study the current implementation that evolved from the original one and is available in Pharo 10.
- The identification of different kinds of pre-conditions: some linked to applicability of the refactorings, some checking possible system breakage and others that are more complex.
- The identification of missed reuse opportunities in current refactorings implementation.

We are well aware that Pharo Refactoring Browser 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 [KBDA16], it has shortcomings. The paper does not aim at criticizing one particular implementation of refactorings 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, the research questions, and the context of this analysis. Section 3 presents an analysis of refactoring pre-conditions.

Section 4 focuses on composition, either explicit or implicit, of refactorings. Section 5 presents a first analysis on how some existing refactorings might be reimplemented as composition of other refactorings. We propose generic guidelines for this work. The paper closes with the related work discussion (Section 6) and the conclusions (Section 7).

## 2. Refactorings and transformations

In this section, we define the domain of our study: We want to understand *whether existing refactorings (behavior preserving modifications of the source code) can be used alongside with behavior agnostic modifications of the source code and possibly share their implementation.* By *behavior agnostic*, we mean that the modification of the source code has no knowledge of (and does not care about) the behavior of this code.

We will first define the vocabulary used in the paper, we then analyze the case of changes of invoked methods[1], with two existing refactorings and a possible transformation. Finally, from this first analysis, we set some research questions.

### 2.1. Vocabulary

We first clarify the vocabulary used in this paper.

**Refactoring:** behavior preserving modification of the source code. Refactorings were introduced by Opdyke [Opd92] and first implemented in Smalltalk by Roberts and Brant [RBJO96, RBJ97, BR98];

**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);

**Pre-condition:** Typically, the implementation of *refactorings* includes some *pre-conditions* that may check the possibility of applying the refactoring. For example, the refactoring Rename Method(oldName,newName) first checks that a newName method does not already exist in the target class;

**Atomic refactoring:** A *refactoring* that is not implemented using some other refactorings (called primitive refactorings in [LT12]);

**Composite refactoring:** A *refactoring* that is implemented using some other refactorings (atomic or composite). This is different from our long term goal which is to be able to express refactorings as a composition of independent pre-conditions and transformations;

**Elementary operation:** A local modification of the source code at one given location, like changing an invocation in a method's body.

[1]Although the refactorings use the phrase 'message send" (e.g.,renameMessageSends, we find it easier to call them method invocations in this paper.

## 2.2. Examples: Changing invoked methods

A source code modification that is often required, is to change the name of an invoked method. This can happen either to rename the method invoked, or to change it for the invocation of another method, or in Smalltalk, to add or remove parameters to the method.

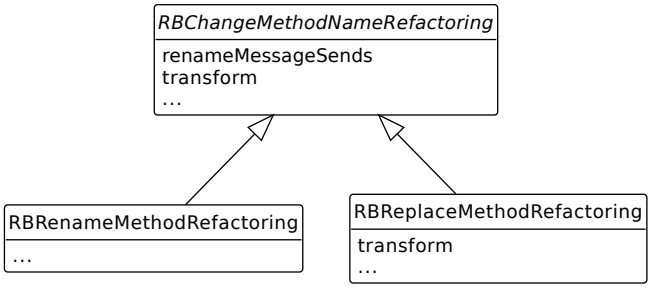


**Figure 1:** The class RBChangeMethodNameRefactoring and two of its subclasses.

In Pharo, these refactorings are implemented by different classes that inherit from the abstract RBChangeMethodNameRefactoring (see Figure 1). Thus, we find RBRenameMethodRefactoring that changes a method's name in the target class and all references (invocations) in the senders of the method.

```
RBChangeMethodNameRefactoring >> transform
  self renameImplementors.
  self renameMessageSends.
  self removeRenamedImplementors
```

Listing 1: RBRenameMethodRefactoring behavior, inherited from RBChangeMethodNameRefactoring.

RBRenameMethodRefactoring renames the implementors with the new name, then renames all the old references, and finally removes the old selector. Its transform method is actually inherited (and not overridden) from its abstract superclass RBChangeMethodNameRefactoring and uses actions defined in this abstract superclass.

There is also another refactoring, Replace Method, implemented by RBReplaceMethodRefactoring, that does not change the name of a method itself but replaces its invocations by invocations to another method.

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

Listing 2: The transformation of the class RBReplaceMethodRefactoring.

This is essentially applying the second step in of the previous refactoring (Listing 1), by using RBChangeMethodNameRefactoring»renameMessageSends defined in the same superclass.

Note that here, since the method that is invoked is not the same after the refactoring (this is not just a change of name), there is no way to guarantee behavior preservation. Thus, this "refactoring" is actually a *transformation*.

This analysis highlights the need for both *transformations* and *refactorings* [SAE+15]. The implementation, based on inheritance, also shows that, from a software engineering point of view, it is important to be able to reuse some subparts of the logic. This situation is emphasized by the definition of new generation refactorings, such as the atomic refactorings supporting live object programming [TPF+18]. We want to understand whether refactorings could be implemented in terms of transformations that would themselves be independent operations, usable by the developers for scripting some development actions.

## 2.3. Research questions

To support the understanding of the duality of refactorings and transformations both at a conceptual and implementation level, this article wants to provide a first answer to the following questions:

- Can refactorings and transformations share their code edition logic?
- Can we decouple code transformations from refactorings to be able to reuse them independently of each other?
- What is the status of the refactoring pre-conditions and their composition?
- Can more complex refactorings be expressed out of simpler ones? How simple/difficult would it be to create a dedicated language of pre-conditions and transformations to express new refactorings?
- What are concrete issues encountered to transform an existing refactoring into a transformation-based refactoring?

This list of questions is only a first step in our more general endeavor.

**Should we turn elementary operation into atomic refactorings?** Some implemented refactorings are directly using elementary operations (such as simply adding an instance variable, removing a method...). For example, adding a method to a class is an elementary operation. But there is also a Add Method refactoring that performs the elementary operation. It accomplishes this by calling the Class»compile: method. Its pre-conditions simply check the applicability of the elementary operation by checking if the name of the new method is available (not already existing in the class). Since many other refactorings are frequently adding new methods, the Add Method Refactoring is one that should be reused by these more complex refactorings.

Therefore, the elementary operations could also be considered atomic refactorings. Such reification would not only be about reuse but also about creating a vocabulary at the same level of abstraction. Still it is unclear if there is a clear win in doing it.

**Can we reuse code editing logic between refactorings and transformations?** Since transformations do not need to preserve behavior, expressing a refactoring as an extension of a transformation would support reuse and avoid logic duplication. This way, refactorings could reuse the transformation logic, only with a few extra checks such as checking behavior conservation pre-conditions.

**What is the status of pre-condition composition in the context of composite refactorings?** Composite refactorings may have their own separate pre-conditions, or only check the ones defined by the invoked refactorings. But there might be a specific time and order to check the invoked refactoring' preconditions. Each invoked refactoring will typically check its own pre-conditions when being executed, but if there are several invoked refactorings, what happens if a later one's pre-conditions are invalidated by a former one's execution?

## 2.4. Context of the analysis

The analysis presented in this article is based on the implementation of Refactorings as done by J. Brant and D. Roberts [RBJO96, RBJ97, BR98] and their evolution as available in Pharo 10 [BDN+09]. Since multiple developers maintained and evolved the original code, our analysis will report a situation that is not the one described in the original document. It may happen that some pre-conditions are missing, or were changed, or that new refactorings are not extending existing ones. In addition, the code is sometimes circonvoluted making the analysis more difficult to perform.

Appendix A presents the list of original refactorings, as described in the PhD of D. Roberts [Rob99]. The Pharo implementation contains more refactorings, as shown in Appendix B.

## 2.5. Implementation overview

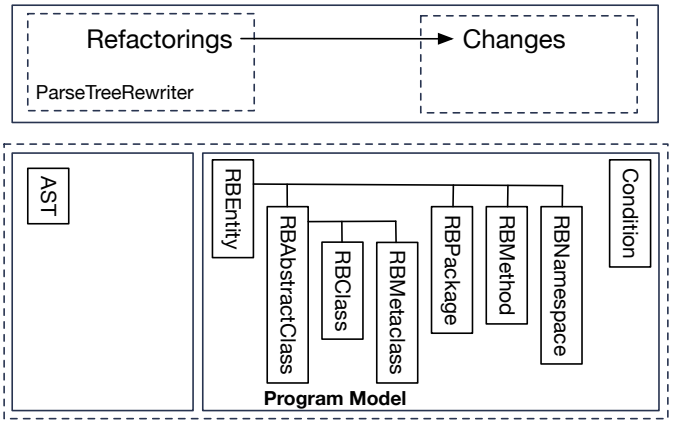


**Figure 2:** Overview of the refactoring engine architecture.

In the Refactoring Browser engine, a refactoring uses a program model to check pre-conditions

and performs code transformations either at the level of the model or using a parse tree rewriter. The output of a refactoring is a sequence of changes that, once applied to the existing code, will perform the refactoring. As shown in Figure 2, the refactoring engine reflects this flow by defining three large elements (dashed boxes):

The *program to be refactored* model (bottom). It is a representation of the program entities (ex: RBMethod, RBClass) and their AST (for methods). The engine does not directly use the reflective language API to perform pre-conditions and the refactorings, it uses its own program model. This allows it to refactor code that does not need to be executable in the current environment itself.

The *refactoring definitions* (top left). This component contains the refactoring definitions. It should be noted that such definitions are entangled with user interactions (such as parameter, new method name request,...) that should be ideally separated from the refactoring itself. This is a factor that can hamper logic reuse.

The *change* model (top right). The change model is a model describing the actions that will be performed. It supports a preview of the refactoring and let the user modify the output of the refactoring. This is the change execution that will actually perform the actual low-level source code editing.

Note that such an architecture supports fast validation of pre-conditions and fast execution of refactorings. According to Kim *et al.,* [KBDA16], the Java refactoring engine is slow because it does not have a model of program other than ASTs.

Many refactorings are using the program model API to perform the code transformations. In terms of refactoring reuse and reification, the question arises whether there is a duplication between the API uses and the refactorings using such API. In the context of our analysis, the following pieces of information are important to assess:

- *Program model API.* Understanding the API of the program model is key, and in particular its use by the implemented refactorings. Indeed, the program model is the lowest API on which pre-conditions are expressed and on which the elementary operations are performed. We want to understand whether such API should be exposed as atomic refactorings or transformations.
- *Reification of elementary operations.* Atomic refactorings directly use sets of elementary operations. As such, they could be seen as a reification of these operations (particularly one an atomic refactoring uses only one elementary operation). We need to assess which (or all) elementary operations can be reified as transformations that would be used by refactorings. In that vision, refactorings could be prohibited to use directly elementary operations.

## 3. Different kinds of pre-conditions

Refactorings have pre-conditions. Such pre-conditions contribute for example to the expression of the behavior preserving aspect of the refactorings, some may also ensure the syntactic or

semantic correctness of the refactorings. We analyzed the pre-conditions of existing refactorings to classify and assess them. A later goal will be to be able to express all refactorings as a clear composition of pre-conditions + transformations, and also to be able to apply transformations without checking the pre-conditions.

Since the implementation of refactorings is using inheritance, some pre-conditions defined in super-classes are shared by several refactorings subclasses. During our analysis, we conceptually flattened such shared pre-conditions to be able to reason about them individually. The analysis shows that (1) there are different families of pre-conditions, and (2) transformations also need pre-conditions, as explained hereafter.

### 3.1. Pre-condition families

In addition to refactorings not defining pre-conditions, our analysis identified three main families of pre-conditions: *Applicability checking*, *Break checking*, *Not idiomatic*. Refactorings with these pre-condition types are listed in Appendix C.

**No pre-condition.** Some refactorings have no pre-conditions. They are listed in Appendix C.1. There is no check for the applicability of these refactorings. Most of them are not defined in the original PhD of D. Roberts, and they probably got added later on by different authors. A deeper analysis is required to assess whether the pre-conditions are not managed at another level (for example in the UI). In addition, some refactorings, such as Extract Set Up Method And Occurrences and Extract Method To Component, are composite refactorings and they "inherit" the pre-conditions from the refactorings they use.

**Applicability check.** The applicability pre-conditions are mainly checking that the refactorings can be applied, *i.e.,* that the *transformation* can be applied (without considering behaviour preservation). The pre-conditions may be checking, for example, that an entity targeted by the refactoring, exists, or that an entity with the same name already exists, or that information (such as names) given is correct. They are listed in Appendix C.2.

The pre-conditions may be expressed as a composition of simple (low-level) conditions implemented as class-side methods in the RBCondition class. Appendix D gives the complete API of this class. The pre-conditions may also be expressed by methods implemented in the program model (see Figure 2).

For example, the following precondition method checks that a class effectively defines a variable before creating its accessors. It uses two methods from RBCondition: #definesClassVariable:in: and #definesInstanceVariable:in:

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

This other pre-condition example uses directly methods from the program model: RBAbstractClass»#hierarchyDefinesInstanceVariable:. It checks, before pulling up an instance variable, that it exists in all the subclasses.

```
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 pull up this variable anyway?' , variableName , '.' ].
    true ]
```

**Break check.** While applicability pre-conditions are related to the existence of a given situation supporting the application of the refactoring (e.g. the method to rename exists), this category express conditions that check whether the application of the refactorings would break the system once applied. So the *applicability check* verify that the transformation can be "physically" applied, and the *break check*, verify that if applied, the system would remain semantically correct.

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. The refactoring checks this by implementing its own precondition methods, that calls a simpler method from RBCondition.

```
RBRemoveClass >> preconditions

  ^ classNames inject: self emptyCondition into: [ :sum :each |
      | aClassOrTrait |
      aClassOrTrait := self model classNamed: each asSymbol.
      aClassOrTrait ifNil: [
        self refactoringFailure: 'No such class or trait' ].
      sum & ((self preconditionIsNotMetaclass: aClassOrTrait)
       & (self preconditionHasNoReferences: each)
       & (self preconditionEmptyOrHasNoSubclasses: aClassOrTrait)
       & (self preconditionHasNoUsers: aClassOrTrait)) ]
```

Listing 3: Remove Class pre-conditions.

**Not idiomatic check** (for lack of a better name). We classified in this family some complex conditions that are implemented in an *ad hoc* way and exhibit some implementation issues. Applicability and break preconditions are mixed with user interaction concerns. Some complex refactorings such as Extract Method, Move Method or Pull Up Method have really complex and large pre-conditions. Section 3.2 presents examples of this.

## 3.2. Complex pre-condition examples

We analyze here two refactorings with complex pre-conditions, such as Move Method and Pull Up Method.

**Move Method.** (see listing below) It moves a method to the class of one of its instance variables. It is a composite refactoring and as such has complex pre-conditions.

```
RBMoveMethodRefactoring >> preconditions
  ^(RBCondition definesSelector: selector in: class)
    & (RBCondition withBlock:
          [self buildParseTree.
          self checkForPrimitiveMethod.
          self checkForSuperReferences.
          self checkAssignmentsToVariable.
          self getClassesToMoveTo.
          self getArgumentNameForSelf.
          self checkTemporaryVariableNames.
          self getNewMethodName.
          true])
```

There are some design flaws in the pre-conditions of this refactoring. A first flaw is that these "conditions" retrieve the class to move the method to (#getClassesToMoveTo), or the new selector of the method to move (#getNewMethodName). It is clear that pre-conditions for this refactoring are not just checking if the refactoring can proceed, but also setting up the transformation since they are in charge of getting additional information.

Another design flaw, is that #getNewMethodName, will present an error dialog to the user in the case of method name collision (when the new method name already exists in the target class). The logic of a refactoring should be independent of the graphical user interface and should not request information from the developer. It should be configured appropriately up front.

These two issues would prevent one from reusing the current refactoring in another, larger, one. They would also impede separating the refactoring in reusable pre-conditions and transformations.

**Pull Up Method.** The Pull Up Method refactoring has some of the most complex pre-conditions. The pre-condition method calls upon several other methods. One of these methods down the chain of calls even performs another refactoring inside the precondition. PullUpMethod»preconditions (Listing 4) calls #PullUpMethod»checkInstVars (Listing 5) which in turn calls #pushUpVariable: (Listing 6), and this last one creates and then executes the refactoring Pull Up Instance Variable.

```
RBPullUpMethod >> preconditions
  self requestSuperClass.
  ^(selectors inject: (RBCondition hasSuperclass: class)
    into: [:cond :each | cond & (RBCondition definesSelector: each in: class)])
      & (RBCondition withBlock:
            [self checkInstVars.
            self checkClassVars.
            self checkSuperclass.
            self checkSuperMessages.
            true])
```

Listing 4: Pull Up Method pre-conditions.

```
RBPullUpMethod >> checkInstVarsFor: aSelector
  class instanceVariableNames do:
```

```
[:each |
((class whichSelectorsReferToInstanceVariable: each) includes: aSelector) ifTrue:
    [ (self confirm: ('<1p> refers to #<2s> which is defined in <3p>. Do you want push up variable #<2s> also
?' expandMacrosWith: aSelector
            with: each
            with: class))
        ifTrue: [ self pushUpVariable: each ]
        ifFalse: [ self refactoringError: 'You are about to push your method without the instance variable it
uses.
        It will bring the system is an inconsistent state. But this may be what you want.
        So do you want to push up anyway?' ] ]]
```

Listing 5: Pull Up Method pre-conditions.

```
RBPullUpMethod >> pushUpVariable: aVariable
  | refactoring |
  refactoring :=  RBPullUpInstanceVariableRefactoring
      model: self model
      variable: aVariable
      class: targetSuperclass.
  self performCompositeRefactoring: refactoring.
```

Listing 6: Pull Up Method calling Pull Up Instance Variable.

### 3.3. Lessons on transformation and pre-conditions

To support the implementation, reuse, and composition of code transformations, it is important to understand the difference between a transformation and a refactoring. As outlined in Section 2.1, an important difference is that a transformation does not have to be behavior preserving (we called it behavior agnostic). At first, we hypothesized that another difference would be that refactorings have pre-conditions while transformations would not need them. There would be a clear dichotomy in refactorings between their pre-conditions on one side and their transformations on the other side, both parts being independent and mutually exclusive.

The analysis of the pre-conditions above shows that not all pre-conditions are concerned with the behavior preserving aspect. For example, we identified the *applicability check* family of pre-conditions. Therefore, we are led to review our initial hypothesis:

- Transformations can have pre-conditions, mainly to check their applicability;
- Among the refactorings, the best candidates to be composed out of transformations are the ones with pre-conditions families *none* and *applicability check*;
- From an implementation point of view, we see that the pre-conditions may be: class-side methods of RBCondition, methods in the program model, or methods in the refactoring class itself. It would seem a good engineering approach to try to standardize these implementations.

**Table 1**
RBEntity Model Operations called by Refactorings.

| **RBAbstractClass** |
|---|
| addInstanceVariable: |
| addMethod: |
| addSubclass: |
| compile: |
| compile:classified: |
| compile:withAttributesFrom: |
| compileTree: |
| convertMethod:using: |
| removeInstanceVariable: |
| removeInstanceVariable:ifAbsent: |
| removeMethod: |
| removeSubclass: |
| renameInstanceVariable:to:around: |
| **RBClass** |
| addClassVariable: |
| addPoolDictionary: |
| addProtocolNamed: |
| comment: |
| removeClassVariable: |
| removeClassVariable:ifAbsent: |
| removePoolDictionary: |
| removeProtocolNamed: |
| renameClassVariable:to:around: |
| **RBMethod** |
| compileTree: |
| **RBNamespace** |
| addClassVariable:to: |
| addInstanceVariable:to: |
| addPackageNamed: |
| addPool:to: |
| addProtocolNamed:in: |
| category:for: |
| changeClass: |
| comment:in: |
| compile:in:classified: |
| convertClasses:select:using: |
| createNewClassFor: |
| createNewPackageFor: |
| defineClass: |
| description: |
| performChange:around: |
| removeClass: |
| removeClassKeepingSubclassesNamed: |
| removeClassNamed: |
| removeClassVariable:from: |
| removeInstanceVariable:from: |
| removeMethod:from: |
| removePackageNamed: |
| removeProtocolNamed:in: |
| renameClass:to:around: |
| renameClassVariable:to:in:around: |
| renameInstanceVariable:to:in:around: |
| renamePackage:to: |
| reparentClasses:to: |
| replaceClassNameIn:to: |

## 4. Refactoring composition analysis

To better understand the current situation, we now analyze existing refactorings, how they are (or not) composed of other refactorings and/or elementary operations. We thus, start by looking at *atomic Refactorings* that are not using other refactorings, although they might be based on elementary operations implemented by a model of the system. Then, we analyze the composite refactorings that do make use of simpler refactorings. Finally, we identify some missed reuse

opportunities: Refactorings that could be calling simpler refactorings, but instead change the program model directly.

## 4.1. Atomic refactorings and operations

As explained in the implementation overview (Section 2.5), refactorings do not modify source code directly, but instead do it through the program model and a number of elementary operations that it offers. This is the API used by every refactoring to apply the changes. It is important to understand this API to identify what operations are available to refactorings or, in the future, to transformations.

Table 1 presents the 53 methods defined in RBEntity subclasses that perform code changes (elementary operations). There are four main subclasses: RBAbstractClass, RBClass, RBMethod, and RBNamespace. All these methods are used by refactorings.

**Table 2**
Atomic refactorings not using other refactorings and how they might be reused by others – An asterisk (*) means that the refactoring adds or removes only one entity in the program model.

| Class | Used By |
|---|---|
| AddClass* | ChildrenToSiblings, CopyClass, SplitClass |
| AddClassVariable* | CopyClass |
| AddInstanceVariable* | CopyClass, SplitClass |
| AddMethod* | CopyClass |
| AddParameter | |
| CategoryRegex | |
| CreateCascade | |
| DeprecateMethod | |
| ExpandReferencedPools | AbstractVariables, PullUpMethod, PushDownMethod |
| ExtractMethod | ExtractMethodAndOccurrences, ExtractMethodToComponent, FindAndReplace |
| InlineMethod | InlineAllSenders |
| InlineParameter | |
| InlineTemporary | |
| ProtocolRegex | |
| RealizeClass | |
| RemoveClass* | |
| RemoveClassVariable* | |
| RemoveInstanceVariable* | SplitClass |
| RemoveMethod | |
| RemoveParameter | |
| RemoveSender | RemoveAllAccessors |
| RenameArgumentOrTemporary | |
| RenameClass | RenamePackage |
| RenameClassVariable | |
| RenameMethod | |
| ReplaceMethod | |
| SourceRegex | |
| SplitCascade | |

Table 2 presents *atomic refactorings*, *e.g.,* the refactorings that are not referencing any other refactorings. *Atomic refactorings* are directly composed of *Elementary operations*.

The table also shows whether these atomic refactorings are reused by other refactorings (*i.e.,* by *composite refactorings* discussed in Section 4.2).

Finally, the table identifies with an asterisk (*) the atomic refactorings that add or remove only one single entity in the program model. These last refactorings correspond to what Santos *et al.,* [SAE+15] defined as "Level one operators": atomic and generic elementary tasks. They are atomic because they describe the addition or deletion of a single code entity. For example, these refactorings are routinely proposed as development helpers (e.g., Add Method). They are generic in the sense that they are independent of the system, the application domain, and sometimes even the programming language.

We analyzed some of the atomic refactorings in Table 2 and will discuss one case here: Add Class refactoring. Listing 7 gives a part of its implementation. It exhibits an opportunity to re-implement the refactoring using a, simpler, transformation.

```
RBAddClassRefactoring >> transform
  self model
    defineClass: ('<1p> subclass: #<2s> instanceVariableNames: '''' classVariableNames: '''' poolDictionaries: ''''
      category: <3p>'
          expandMacrosWith: superclass
          with: className
          with: category asString);
    reparentClasses: subclasses to: (self model classNamed: className asSymbol)
```

Listing 7: AddClassRefactoring

We see in the last line of Listing 7 that the refactoring can actually support the insertion of a class within a hierarchy. If the proposed parent of the new class has no subclasses then the new class is created and nothing else happens. But this refactoring can accept subclasses of the parent class as parameters to insert the new class between their superclass (to become parent of the new class) and themselves. This is done by the #reparentClasses:to: call at the end of the listing. This refactoring could be called Insert Class and could invoke a transformation Add Class that would only perform a class addition. This example raises also the question of the customization of the refactoring logic and whether such customization should be promoted as first class citizen or not.

## 4.2. Explicit composite refactorings

To understand how to compose refactorings, we analyzed *composite refactorings* that explicitly refer to other refactorings in their implementation. Appendix E presents all the composite refactoring we found. It is, in a sense, the counterpart of Table 2 (showing atomic refactorings used in composite ones).

For example, both Pull Up Method and Push Down Method are composite refactorings. They both invoke the Expand Referenced Pools refactoring.

Pull Up Method is a refactoring for moving methods up in the inheritance hierarchy from subclasses to their superclass. Implementation details for this refactoring are shown in Listing 8. Before recompiling the target method in the superclass (last statement), another

refactoring is executed: Expand Referenced Pools. Listing 8 shows that the last operation (compile:classified:) that this refactoring performs is the same change as Add Method. Add Method also uses RBAbstractClass»compile:classified: in the same manner. This presents an opportunity to reuse Add Method instead of redefining it separately.

```
RBPullUpMethodRefactoring >> pullUp: aSelector
  | source refactoring |
  source := class sourceCodeFor: aSelector.
  source ifNil: [self refactoringFailure: 'Source for method not available'].
  refactoring := RBExpandReferencedPoolsRefactoring
        model: self model
        forMethod: (class parseTreeFor: aSelector)
        fromClass: class
        toClasses: (Array with: targetSuperclass).
  self performCompositeRefactoring: refactoring.
  targetSuperclass
    compile: source
    classified: (class protocolsFor: aSelector)
```

Listing 8: Pull Up Method implementation core.

Push Down Method refactoring, conversely, moves methods down the inheritance hierarchy. It has a similar implementation with a #pushDown: method that resembles the #pullUp: method, also using the Expand Referenced Pools refactoring before recompiling the target method in each subclass.

### 4.3. Implicit composite refactorings: Missed reuse opportunity

Implicit composite refactorings are those that could be calling simpler refactorings but instead change the model directly, duplicating functionalities implemented elsewhere. They should probably be changed to reuse these other refactorings and become explicitly composite. In this article we do not verify this hypothesis because it would imply a large development effort. This is something that we plan to do latter.

We list the implicit composite refactorings that we identified in Table 3, and Table 4 proposes some potential reuses in these implicit composite refactorings.

For example, Pull Up Instance Variable does not use Add Instance Variable, and its pre-conditions do not take into account the pre-conditions of Add Instance Variable (which is checking if a name is valid, and if a variable with the same name does not already exist in the hierarchy). Pull Up Instance Variable is based on the assumption that the variable to be pulled up already satisfies the validity constraints. It checks that the instance variable to be pulled up is defined in the hierarchy. We believe that not checking name validity is an optimization that is not worth the risk it introduces. Reusing the logic of Add Instance Variable would make sure that all the names are validated.

## 5. Potential transformation candidates

In this section, we build on the analyses that were presented in this paper to propose some refactorings that could be turned into transformations and thus be reused by the original or other

**Table 3**
Implicitly composite refactorings: composite refactorings not reusing existing atomic ones.

- ChangeMethodNameRefactoring
- ChildrenToSiblings
- ClassRegex
- CreateAccessorsForVariable
- CreateAccessorsWithLazyInitializationForVariable
- DeprecateClass
- GenerateEqualHash
- GeneratePrintString
- MoveInstVarToClass
- MoveMethodToClass
- PullUpClassVariable
- PullUpInstanceVariable
- PullUpMethod
- PushDownClassVariable
- PushDownInstanceVariable
- RemoveClassKeepingSubclasses
- RemoveMethod
- RenameInstanceVariable
- SwapMethod
- TemporaryToInstanceVariable

refactorings. They could also be used directly by any developer ready to take the responsibility to automatically modify the source code without the security of behavior preservation.

## 5.1. Reusing Add Method

In a previous example, Push Down Method refactoring was shown to be a composite refactoring because it invokes Expand Referenced Pools (Section 4.2). We also said that it could use Add Method, since its last statement does the same thing. Here we propose a new AddMethodTransformation (9) and a modification of RBPushDownMethodRefactoring to use this transformation (Listing 10).

The new transformation only compiles the source code of the method to be added in its parent class.

```
RBAddMethodTransformation >> transform
  class compile: source classified: protocols
```

Listing 9: Proposed RBAddMethodTransformation

Push Down Method could be altered in the following way and maintain its behavior. It can be compared to the very similar Pull Up Method refactoring of Listing 8, where the important difference occurs last (in bold), when applying the transformation.

```
RBPushDownMethodRefactoring >> pushDown: aSelector
  | code protocols refactoring addMethodRef|
  code := class sourceCodeFor: aSelector.
  protocols := class protocolsFor: aSelector.
  refactoring := RBExpandReferencedPoolsRefactoring
        model: self model
        forMethod: (class parseTreeFor: aSelector)
        fromClass: class
        toClasses: self allClasses.
  self performCompositeRefactoring: refactoring.
```

**Table 4**
Potential reuse in implicit composite refactorings

| **Refactoring** | could use… | …instead of |
|---|---|---|
| ChangeMethodName | RemoveMethod | removeMethod: |
| CreateAccessorsForVariable | AddMethod | compile: |
| CreateAccessorsWithLazy-InitializationForVariable | AddMethod | compile: |
| DeprecateClass | AddMethod | compile: |
| GenerateEqualHash | AddMethod | compile: |
| GeneratePrintString | AddMethod | compile: |
| MoveInstVarToClass | AddMethod<br>RemoveInstanceVariable | addMethod: and compile:<br>removeInstanceVariable |
| MoveMethodToClass | AddMethod<br>RemoveMethod | addMethod: and compile:<br>removeMethod: |
| PullUpClassVariable | RemoveClassVariable | |
| PullUpInstanceVariable | RemoveInstanceVariable | |
| PushDownClassVariable | RemoveClassVariable | |
| PushDownInstanceVariable | RemoveInstanceVariable | |
| RemoveMethod | RemoveMethod | |
| SwapMethod | AddMethod and<br>RemoveMethod | compile: and<br>removeMethod: |
| TemporaryToInstanceVariable | RemoveInstanceVariable | |

```
self allClasses do: [ :each |
  (each directlyDefinesMethod: aSelector) ifFalse: [
    addMethodRef := AddMethodTransformation
      model: self model
      addMethod: code
      toClass: each
      inProtocols: protocols.
    self performCompositeRefactoring: addMethodRef ] ]
```

Listing 10: Proposed RBPushDownMethodRefactoring using the RBAddMethodTransformation

While this specific example ends up with more code, it would be easier to maintain in case the program model's API were to change. Instead of looking for every instance where the model's API is invoked, only Add Method would need to be modified.

## 5.2. Rename Method and Replace Method Revisited

In Section 2.2 we discussed the implementation of the Replace Method and Rename Method refactorings.

Rename method is one of the most used refactorings. It boils down to the following steps as described in Fowler's book [FBB+99]

1. Check that a newName method does not exist in the class and its superclass;

2. Add a new method with same body than the old method but with the newName;
3. Identify all call sites of the oldName method and rewrite them to invoke newName;
4. Remove the method oldName from the class.

Replace Method(name1, name2) replaces a method's invocations by invocations to another method:

1. optional: Check that there is a method name1 in the system.
2. Identify all the call sites of the name1 method and rewrite them to invoke the name2 method.

Clearly, Replace Method cannot guarantee behavior preservation if the new method does something different from the old one. As such it is not an actual refactoring and would rather be named RBReplaceMethodTransformation and exposed to the user as a transformation. It would be more indicative of its behavior agnostic nature.

We saw previously, in Listing 2, the core logic of this transformation (still called a refactoring at that time). It calls RBReplaceMethodNameTransformation»renameMessageSends (see Listing below) to converts all references to the old method name. This later transformation is also used in Rename Method refactoring as one of three operations (see Listing 1).

```
RBReplaceMethodNameTransformation >> renameMessageSends
  self convertAllReferencesTo: oldSelector using: self parseTreeRewriter
```

### 5.3. Discussion

The analyses performed and reported in previous sections suggest actions to improve the reuse of logic between transformations and refactorings. Our goal being the crisp identification of the concept of *transformation*, there is a clear need to identify which transformations should be proposed to the developers. As identified in the previous sections, transformations could emerge from the current refactorings or from the current elementary operations. The vocabulary that the set of transformations will offer must not be defined abstractly as it could lead to an artificial API not fitting the needs of developers. We started to analyze the implementation of the refactorings and the use of the elementary operations but more should be done. We sketch here some general guidelines that future work will have to validate.

- Refactorings having only *applicability check* pre-conditions are good candidate to become transformations. They do not consider behavior preservation and as such can be used as transformation;
- For refactorings having only *break check* pre-conditions, whether they could be turned into transformations by ignoring their precondition should be further investigated. It is also unclear whether extracting the transformation into first class entities (that can be called by the refactoring) is the right way to go;

- Low-level APIs from *program model* used by refactorings show opportunities to create and call a corresponding transformation (*i.e.,* create a transformation for each *elementary operation*). Now the question whether *all* of the low-level APIs are interesting from the point of view of a developer wanting to apply transformations is still raised. A deeper analysis of the API is needed;

- There is a definitive need to separate the user interaction logic from the actual refactoring. We expect that such a separation would support better reuse of logic between refactorings and transformations;

- We asked ourselves whether a developer in need to script code changes should be exposed to refactorings or transformations. As proposed in the first point, our conclusion is that for refactorings with only applicability checking pre-conditions, the difference does not matter so using the term refactorings may be the least disturbing.

  For the other refactorings, having the choice between a transformation or a refactoring is important because some transformations cannot preserve behavior. From this analysis we see that a library should offer then both, with some refactorings offering a transformation counterpart (being it or not reused internally by the refactoring);

- The non-derivation of the pre-conditions of composite refactorings proposed by Li and Thomson [LT12] is interesting. They propose to not try to derive the pre-conditions of composed refactorings from all the pre-conditions of their components, but rather to apply each component independently of the others with its own pre-conditions.

  We agree with this view as refactorings first build a change model on top of which subsequent pre-conditions could be validated. This also supports the possibility to roll-back composite refactorings.

## 6. Related work

There is really little research focused on the engineering and definitions of refactorings themselves.

**Independent and cross languages.** While the definition of language independent or cross language refactorings does not focus the reuse of transformation logic, they are the only work beside the PhD of D. Roberts formalizing refactoring implementation. Tichelaar [TDDN00, Tic01] presents some language independent refactorings on top of the FAMIX metamodel [DAB+11] while Mayer *et al.,* presents a metamodel to support cross language refactorings [MSL12]. Such approaches are interesting because they focus on the implementation of the refactorings. Nevertheless, they do not provide an analysis on the reuse of transformation and composition of refactorings.

**Refactoring engines.** There is some work on refactoring engines for languages such as Erlang with tidier [SA09, AS09], Wrangler [LTOT08], or refactoring for Ruby (RubyMine from

jetbrains). But there is no explanation or information about the actual implementation of the refactoring engines themselves. Refactorings are simply explained from a user perspective.

Kim *et al.,* [KBDA16] 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 in fact similar to the one of the Refactoring browser in Pharo that we studied in this paper. It is similar in the sense that R3 uses an in-memory model of program and is not exclusively manipulating ASTs. R3 model is a kind of direct database schema with foreign keys. In addition, it contains information about the program entities encoded as boolean.

**Code to code transformation.** Some work such as in [KWK04, Cin01] focuses on the derivation of composite refactoring precondition from its constituents. The idea is to be able to execute the composite preconditions before performing the actual code transformation. Li and Thomson [LT12] presents a DSL to define new refactorings in Erlang for Erlang. This is one of the rare article discussing the notion of atomic and non-atomic composite refactorings. With Wrangler a primitive refactoring is extended with a refactoring command generator. During composition, and as a design choice, Wrangler does not do anything to derive composite refactoring precondition instead each primitive refactorings is executed individually. Hills *et al.* [HKV12] show how the authors integrate Eclipse and Rascal to be able to perform a transformation from a visitor to an interpreter design pattern. They use Rascal as a meta programming environment.

**User and usabilities.** Boshernotan *et al.* [BG04, BGH07] propose a program manipulation paradigm that enables programmers to change source code with interactively-constructed visual program transformations. Similarly Rizun *et al.* [RBD15] propose direct manipulation of AST nodes to generate corresponding code transformations using Refactoring Parse Tree Rewriter [BR98].

**Semantics-driven.** Kesseli, in his PhD [Kes18], explores semantics-driven refactorings by 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 [FTSC12]), general refactorings [TME+18]) with tools such as RefactoringMiner2.0. Other publications mine missed opportunities to refactor code (move method [TC09], missed polymorphism [TC10].) The work presented in this article is concerned about the implementation and in particular the reuse of logic between transformations and refactorings — not the applications of refactorings on existing code base.

## 7. Conclusion

The goal of this article is to understand whether it would make sense to compose refactorings out of pre-conditions, and simple code transformations or other refactorings. For this we did a deep analysis of the current implementation of the Refactoring Browser as defined by D. Roberts and J. Brant.

We presented a classification of pre-conditions that identified four families of pre-conditions. In particular, we learned that some pre-conditions are mainly checking the applicability of their refactoring and that, as such, a corresponding transformations would benefit from having the same pre-conditions.

We studied atomic refactorings (the ones that do not reuse other refactorings) as well as the elementary operations offered by the system to actually realize the code changes. We wanted to understand why all elementary operations are not exposed as transformation (or refactorings), and whether this prevented possible cases of refactoring reuse instead of using the elementary operations. It is still unclear whether the reification of all elementary operations would be a real benefit or would just be a case of over engineering.

Furthermore, we studied explicit composite refactorings to understand how the pre-conditions and the atomic refactorings were inter-playing. This led to the analysis of implicit composited refactorings (refactorings compose of multiple transformations but not through other refactorings, i.e. re-implementing other refactoring transformations). We show that some implicit composite refactorings can easily be turned into explicit ones.

Our analysis is the first step toward the design of co-existing and collaborating transformations and refactorings. The next step will be to put our recommendatin in practice, like decouple the user interaction from the refactorings and actually convert implicit composite refactorings into explicit ones.

## References


[AS09] Thanassis Avgerinos and Konstantinos Sagonas. Cleaning up erlang code is a dirty job but somebody's gotta do it. In Clara Benac Earle and Simon J. Thompson, editors, *8th Workshop on Erlang*, pages 1–10. ACM, 2009.

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[BG04] M. Boshernitsan and S. L. Graham. ixj: Interactive source-to-source transformations for java. In *OOPSLA Companion*, 2004.

[BGH07] Marat Boshernitsan, L. Susan Graham, and A. Marti Hearst. Aligning development tools with the way programmers think about code changes. In *Conference on Human Factors in Computing Systems (CHI '07)*, April 2007.

[BR98] John Brant and Don Roberts. "Good Enough" Analysis for Refactoring. In *Object-Oriented Technology Ecoop '98 Workshop Reader*, LNCS, pages 81–82. Springer-Verlag, 1998.

[Cin01] Mel O Cinnéide. *Automated application of design patterns : a refactoring approach*. PhD thesis, Trinity College - School of Computer Science and Statistics, 2001.

[DAB+11] Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family. Technical report, RMod – INRIA Lille-Nord Europe, 2011.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[FTSC12] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Identification and application of Extract Class refactorings in object-oriented systems. *Journal of Systems and Software*, 85(10):2241–2260, October 2012.

[GDMH12] Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. Reconciling manual and automatic refactoring. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 211–221, Piscataway, NJ, USA, 2012. IEEE Press.

[HKV12] Mark Hills, Paul Klint, and Jurgen J. Vinju. Scripting a refactoring with rascal and eclipse. In *5th Workshop on Refactoring Tools*, pages 40–49, 2012.

[KBD15] Jongwook Kim, Don Batory, and Danny Dig. Scripting parametric refactorings in java to retrofit design patterns. In *ICSME*, 2015.

[KBDA16] Jongwook Kim, Don Batory, Danny Dig, and Maider Azanza. Improving refactoring speed by 10x. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1145 – 1156, 2016.

[Kes18] Pascal Kesseli. *Semantic Refactorings*. PhD thesis, University of Oxford, 2018.

[KWK04] Günter Kniesel-Wünsche and Helge Koch. Static composition of refactorings. *Science of Computer Programming*, 52:9–51, August 2004.

[LT12] Huiqing Li and Simon Thompson. A domain-specific language for scripting refactorings in erlang. In *FASE*, 2012.

[LTOT08] Huiqing Li, Simon Thompson, György Orosz, and Melinda Tóth. Refactoring with wrangler, updated: Data and process refactorings, and integration with eclipse. In *Workshop on ERLANG*, pages 61–72, New York, NY, USA, 2008. Association for Computing Machinery.

[MHPB11] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2011.

[MSL12] Philip Mayer, Andreas Schroeder, and Welf Löwe. Cross-language code analysis and refactoring. In *In Proceedings of the International Workshop on Source Code Analysis and Manipulation*, 2012.

[NCV+13] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. A comparative study of manual and automated refactorings. In *27th European Conference on Object-Oriented Programming*, pages 552–576, 2013.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[RBD15] Markiyan Rizun, Jean-Christophe Bach, and Stéphane Ducasse. Code transformation by direct transformation of asts. In *International Workshop on Smalltalk Technologies (IWST)*, 2015.

[RBJ97] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[RBJO96] Don Roberts, John Brant, Ralph E. Johnson, and Bill Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96*, April 1996.

[Rob99] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999.

[SA09] Konstantinos Sagonas and Thanassis Avgerinos. Automatic refactoring of erlang programs. In António Porto and Francisco Javier López-Fraguas, editors, *International Conference on Principles and Practice of Declarative Programming*, pages 13–24. ACM, 2009.

[SAE+15] Gustavo Santos, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Túlio Valente. System specific, source code transformations. In *31st IEEE International Conference on Software Maintenance and Evolution*, pages 221–230, 2015.

[SvP12] F. Steimann and J. von Pilgrim. Constraint-based refactoring with foresight. In *ECOOP*, 2012.

[TC09] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, 2009.

[TC10] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of refactoring opportunities introducing polymorphism. *Journal of Systems and Software*, 83(3):391–404, 2010.

[TDDN00] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings of International Symposium on Principles of Software Evolution*, ISPSE'00, pages 157–167. IEEE Computer Society Press, 2000.

[Tic01] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, December 2001.

[TME+18] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*, pages 483–494, New York, NY, USA, 2018. ACM.

[TPF+18] Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, and Stéphane Ducasse. Dynamic software update from development to production. *Journal of Object Technology*, 17:1–36, 2018.

[VCM+13] M. Vakilian, N. Chen, R. Z. Moghaddam, S. Negara, and R. E. Johnson. A compositional paradigm of automating refactorings. In *European Conference on Object-Oriented Programming*, pages 527–551, 2013.

[VCN+12] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. Use, disuse, and misuse of automated refactorings. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 233–243, Piscataway, NJ, USA, 2012. IEEE Press.

[VEdM06] M. Verbaere, R. Ettinger, and O. de Moor. Jungl: a scripting language for refactoring. In *Proceedings of International Conference on Software Engineering*, 2006.

## A. Original list of refactorings

Original list of refactorings as in [Rob99] 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
- Rename Method
- Rename Temporary

## B. Refactorings added in Pharo

- Abstract Variables
- Accessor Class
- Add Method
- 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

## C. Refactorings with their pre-condition families

We found four pre-condition families in the refactorings: None, Applicability check, Break check, and Not idiomatic. These families were described in Section 3. We list here in alphabetical order the refactorings using each family of pre-condition.

### C.1. Refactorings with pre-condition family: None

- Abstract Class Variable
- Abstract Variables
- Category Regex
- Class Regex
- Extract Method To Component
- Extract Set Up Method And Occurrences
- Expand Referenced Pools
- Protocol Regex
- Source Regex

### C.2. Refactorings with pre-condition family: Applicability check

- Abstract Instance Variable
- Accessor Class
- Add Class
- Add Class Variable
- Add Instance Variable
- Add Method
- Add Parameter
- Children To Siblings
- Copy Class
- Copy Package
- Create Accessors For Variable
- Create Cascade
- Create Lazy Initialization
- Deprecate Class
- Deprecate Method
- Extract Method And Occurrences
- Extract Set Up Method
- Extract To Temporary
- Find And Replace
- Find And Replace Set Up

- Generate Equal Hash
- Generate Print String
- Inline All Senders
- Inline Method
- Inline Parameter
- Merge Instance Variable Into Another
- Move Inst Variable To Class
- Move Method To Class
- Move Method To Class Side
- Move Variable Definition
- Protect Instance Variable
- Pull Up Class Variable
- Pull Up Instance Variable
- Push Down Method
- Realize Class
- Remove All Senders
- Remove Hierarchy Method
- Rename Argument Or Temporary
- Rename Class
- Rename Class Variable
- Rename Instance Variable
- Rename Method
- Rename Package
- Replace Method
- Split Cascade
- Split Class
- Swap Method

### C.3. Refactorings with pre-condition family: Break check

- Remove Parameter
- Remove Sender
- Inline Method From Component
- Remove Class Variable
- Push Down Instance Variable
- Remove Instance Variable
- Temporary To Instance Variable
- Remove Class
- Remove Class Keeping Subclasses
- Push Down Class Variable

### C.4. Refactorings with pre-condition family: Not idiomatic

- Extract Method
- Move Method
- Remove Method
- Pull Up Method

## D. API of the RBCondition class

List of simple (low-level) conditions implemented in the RBCondition class. These methods are used to create more complex pre-conditions as described in Section 3.

- #accessesClassVariable:in:showIn:
- #accessesInstanceVariable:in:showIn:
- #canUnderstand:in: checkClassVarName:in:
- #checkInstanceVariableName:in:
- #checkMethodName:
- #checkMethodName:in:

- #definesClassVariable:in:
- #definesInstanceVariable:in:
- #definesSelector:in:
- #definesSelector:in:orIsSimilarTo:
- #definesTempVar:in:ignoreClass:
- #definesTemporaryVariable:in:
- #directlyDefinesClassVariable:in:
- #directlyDefinesInstanceVariable:in:
- #hasSubclasses:
- #hasSubclasses:excluding:
- #hasSuperclass:
- #hierarchyOf:canUnderstand:
- #hierarchyOf:definesVariable:
- #hierarchyOf:referencesInstanceVariable:
- #isAbstractClass: isClass:
- #isEmptyClass:
- #isGlobal:in:
- #isImmediateSubclass:of:
- #isMetaclass:
- #isSubclass:of:
- #isSymbol:
- #isValidClassName:
- #isValidClassVarName:for:
- #isValidInstanceVariableName:for:
- #isValidMethodName:for:
- #methodDefiningTemporary:in:ignore:
- #referencesInstanceVariable:in:
- #reservedNames
- #subclassesOf:referToSelector:
- #validClassName:
- #withBlock:

## E. Explicit Composite refactorings in Pharo

We list here the explicit composite refactorings that we found in Pharo and the simpler refactorings they are composed of.

| Refactoring | Uses refactorings |
|---|---|
| AbstractClassVariable | CreateAccessorsForVariable |
| AbstractInstanceVariable | CreateAccessorsForVariable |
| AbstractVariables | CreateAccessorsForVariable, ExpandReferencedPools |
| AccessorClass | CreateAccessorsForVariable |
| ChildrenToSiblings | AddClass, PullUpInstanceVariable, PullUpClassVariable |
| CopyClass | AddClass, AddMethod, AddInstanceVariable, AddClassVariable |
| CopyPackage | CopyClass |
| ExtractMethodAndOccurrences | ExtractMethod, FindAndReplace |
| ExtractMethodToComponent | ExtractMethod, InlineAllSenders, MoveMethod |
| ExtractSetUpMethodAndOccurrences | FindAndReplaceSetUp, ExtractSetUpMethod |
| ExtractSetUpMethod | TemporaryToInstanceVariable |
| FindAndReplace | ExtractMethod |
| FindAndReplaceSetUp | ExtractSetUpMethod |
| InlineAllSenders | InlineMethod, RemoveMethod |
| InlineMethodFromComponent | AbstractVariables |
| MergeInstanceVariableIntoAnother | CreateAccessorsForVariable |
| MoveMethod | AbstractVariables |
| MoveMethodToClassSide | CreateAccessorsForVariable |
| ProtectInstanceVariable | InlineAllSenders |
| PullUpMethod | ExpandReferencedPools, PullUpInstanceVariable |
| PushDownMethod | ExpandReferencedPools |
| RemoveAllSenders | RemoveSender |
| RemoveClassKeepingSubclasses | PushDownClassVariable, PushDownInstanceVariable, PushDownMethod |
| RemoveHierarchyMethod | RemoveMethod |
| RenameInstanceVariable | CreateAccessorsForVariable |
| RenamePackage | RenameClass |
| SplitClass | AddClass, AddInstanceVariable, CreateAccessorsForVariable, RemoveInstanceVariable |