Transformation-based Refactorings: a First Analysis

N. Anquetil, M. Campero, S. Ducasse, J.-P. Sandoval and P. Tesone

Abstract

Refactorings are well-known behavior preserving transformations. Little work exists on the analysis of their implementation and in particular how ~~traditional~~ refactorings might be composed from smaller, reusable, parts. In this article we study the seminal implementation and evolution of Refactorings as proposed in the PhD of D. Roberts and that it implemented in the Refactoring Browser package in Pharo. In particular we focus on the possibilities to reuse transformations independently from the behavior preserving aspect of refactoring. The question we want to answer is: Is it possible to have more atomic transformations and refactorings composed ~~out~~ of such transformations? We study the expressed preconditions of existing refactorings and identify several families. We identify missed opportunities of reuse in the case of implicit composite refactorings. This analysis should the basis for composable refactorings and tool assisted transformations the do not aim at behavior preservation.

1. Introduction

Refactorings are behavior preserving code transformations. The seminal work of Opdyke [1] and the Refactorings Browser (first implementation of Refactorings of Roberts and Brant [2, 3, 4]) paved the way to the spread of refactorings [5]. A plethora of research has been performed on refactorings such as for their detection [6], practioner use [7, 8, 9, 10], or atomic refactorings for live environments [11]. Refactorings are now a must-have standard in modern IDEs [12, 9, 8, 10, 13].

~~Still~~ from a daily development perspective, refactorings and their behavior preserving form are not enough [14]. Non behavior preserving code transformations are also needed. 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 well not preserve behavior, yet it is a need that arises in real situations. It is clear that Replace Call has strong similarities with the Rename Method refactoring, but it would be awkward for a developer to perform it by applying the refactoring. When in need for this 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 [14].

Defining some specific code transformations such as Replace Call is our long term engineering goal. While ~~targeting~~ this goal, we wish to shed a new light on the following related questions:

- Can both refactorings and ~~transformations~~ share their code transformation logic?
- Can we decouple code transformations from refactorings to ~~be able to~~ reuse them when behavior preservation is not a concern?

IWST

- Can we compose refactorings from such code transformations?
- Can complex refactorings be expressed out of simpler ones?
- What would be the impact on the Refactoring preconditions?
- How would preconditions of complex refactorings compose?

This article ~~is~~ focus~~ing~~ on the duality of preconditions and transformations that make refactorings, and how they can be composed together. Our contributions are the following:

- An analysis of the original implementation of Refactorings. We study the current implementation that evolves from the original one and is available in Pharo 10.
- The identification of different kinds of preconditions: 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.

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 preconditions. Section 4 focuses on composition either explicit or implicit of refactorings. Section 4 presents paths to identify and reuse transformations.

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 refer to the etymological sense[1], meaning 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, 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 [1] and first implemented in Smalltalk by Roberts and Brant [2, 3, 4];

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);

---

[1] *a*=not/without, *gnōstos*=know

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;

Elementary refactoring: A *refactoring* that is not implemented using some other refactorings;

Composite refactoring: A *refactoring* that is implemented using some other refactorings (elementary or composite);

Elementary operation: A local modification of the source code, like changing an invocation in a method’s body.

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.

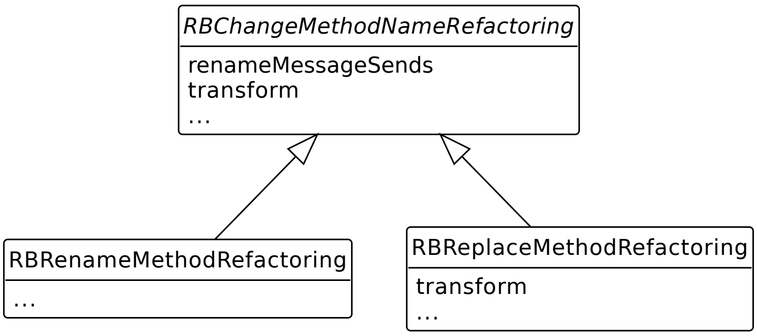
RBChangeMethodNameRefactoring
renameMessageSends
transform
...
RBRenameMethodRefactoring
...
RBReplaceMethodRefactoring
transform
...

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

In Pharo, these refactorings are implemented by different classes that all inherit from the abstract RBChangeMethodNameRefactoring (see also 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.

~~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.

The refactoring renames the implementors with the new name, 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 invocation 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 only applying the second step in of the previous refactoring (Listing 1), again 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* [14]. 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 [11].~~ We want to understand whether refactorings could be implemented in terms of transformations that would themselves be independent operations, usable by the developers.

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 transformation logic?~~
- ~~Can we decouple code transformations from refactorings to be able to reuse them independently of each other?~~
- ~~What is the impact on the refactoring preconditions?~~
- ~~Can more complex refactorings be expressed out of simpler ones? What is the status of preconditions and their composition?~~
- ~~What are concrete issues encountered to transform an existing refactoring into a transformation-based refactoring?~~

What is the cost to introduce a transformation? Some old refactorings are directly accessing elementary operations.

Adding a method to a class is something that could be considered an elementary operation. Elementary operations are those that simply add or remove one element of an object. This could be adding or removing a class, a method, or a variable. ADD METHOD is a refactoring that does one very elementary action, which is to add a single method to a given class. This refactoring accomplishes this by calling the Class»compile: method. Its preconditions simply check ~~the applicability of this operation by checking~~ if the name for this new method is available. Since many other refactorings are frequently adding new methods, the ADD METHOD Refactoring is one that should be reused by these more complex refactorings.

Can we reuse transformation logic between refactorings and transformations? ~~Since~~ transformations do not need to preserve behavior, it would be easier to add some procedures to a transformation to make them refactorings. ~~This way~~ the refactorings can reuse the transformation logic, only with a few extra checks such as break checking preconditions.

What is the status of precondition composition in the context of composite refactorings? Composite refactorings check the preconditions of any refactorings they invoke when using RBRefactoring»performCompositeRefactoring. They may have their own separate preconditions or only check the ones defined by the invoked refactorings. Each invoked refactoring checks its preconditions at the time of being executed, so the first precondition check is that of the composite refactoring, and then once it is executing its transformation it will check the preconditions of the refactorings it is using. Since the precondition check is done before the transformation, it would be possible for one of the inner refactorings to invalidate the composite refactoring's preconditions after it is executed.

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 [2, 3, 4] and their evolution as available in Pharo [15]. ~~Caveat, since~~ multiple developers maintained and evolved the code, our analysis will report a situation that is not the one described in the original document. It may happen that some preconditions are missing or where changed or that new refactorings are not extending existing ones.

Appendix A presents the list of original refactorings as described in the PhD of D. Roberts [16]. The Pharo implementation contains more refactorings as shown in Appendix B.

2.5. Implementation overview

As shown in Figure 2, the refactoring engine is defined with three large elements (dashed boxes): the *program* model (bottom) with a representation of the entities (ex: RBMethod, RBClass) and their AST (for methods); *refactoring definitions* (top left), and *change* model (top right). In essence a refactoring uses a program model to check preconditions 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.

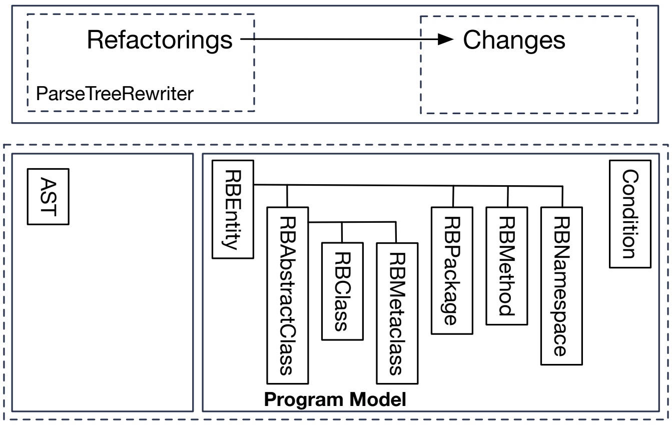

Figure 2: Overview of the refactoring engine architecture.

Many refactorings are using the program model API to perform the code transformations. In terms of refactoring reuse and reification, the question to understand 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:

- *API.* Understanding the API of the program model is key and in particular its use by the actual refactorings. Indeed the program model is the lowest API on which preconditions are expressed and on which the actual program modifications are performed. We want to understand whether such API should be exposed as Elementary refactorings or transformations.
- *Reification as elementary operations.* An elementary refactoring reifies a set of elementary operations, however it is unclear if the necessary API is reifed and reused by refactorings. In addition, we need to assess whether an elementary operation can be reified as a first class transformation that can be reused by refactorings.

3. Different Kinds of Preconditions

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

3.1. Precondition families

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

No precondition. Some refactorings have no preconditions. 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 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, such as Extract Set Up Method And Occurrences and Extract Method To Component, are composite refactorings and they "inherit" the preconditions from the refactorings they use.

Applicability check. The applicability preconditions are mainly checking that the refactorings can be applied. They are listed in Appendix C.2. The pre-conditions may be checking, for example, that an entity, target of the refactoring, exists, that an entity with the same name already exists, or that information (such as names) given is correct.

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 from methods implemented in the program model.

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 precondition example uses directly methods from the program model: RBAbstractClass»#hierarchyDefinesInstanceVariable:. It checks, before pulling up an instance variable, that it exists in the all the sub-classes.

```
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 ]
```

Listing 3: Pull Up Instance Variable preconditions.

Break check. While applicability preconditions are related to the existence of a given situation supporting the application of the refactoring, this category uses preconditions that check whether the application of the refactorings would break the system.

For example the Remove Class refactoring checks that the class is not referenced anymore that it does not have subclasses, that it is not used by other classes or that it is not a metaclass. The refactoring checks this by implementing its on pre-condition methods, that call 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 4: Remove Class preconditions.

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. We give some examples in next section. Some complex refactorings such as Extract Method, Move Method or Pull Up Method has really complex and large preconditions. Section 3.2 presents one example.

3.2. Complex precondition examples

We analyze here two refactorings with complex preconditions such as Move Method and Pull Up Method.

Move Method (see Listing 5). It moves a method to the class of one its instance variables. It is a composite refactorings and as such as also complex preconditions.

```
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])
```

Listing 5: Move Method

There are some design flaws in the preconditions of this refactoring. First, these "preconditions" retrieve the class to move the method to (#getClassesToMoveTo), or the new selector of the method to move (#getNewMethodName). It is clear that "preconditions" 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 a method like #getNewMethodName, in the case of method name collision (when the new method name already exists in the target class), will present an error dialog to the user and ask if he wishes to change the name. The logic of a refactoring should be independent from the graphical user interface and should not request information from the developer. It should be configured appropriately up front. This is a key point if we want to be able to reuse the refactorings, for example by composing them. It is also important if we want to separate refactorings in reusable preconditions and tranformations.

Pull Up Method. The Pull Up Method refactoring is a unique refactoring in the sense that it has some of the most complex preconditions. The preconditions method calls upon several other methods of its own. One of these methods down the chain of calls even performs another refactoring of its own (Pull Up Method is composed of another refactoring). PullUpMethod»preconditions (Listing 6) calls #PullUpMethod»checkInstVars which in turn calls #pushUpVariable: (Listing 7), 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 6: Pull Up Method Preconditions.

```
RBPullUpMethod >> pushUpVariable: aVariable

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

Listing 7: Pull Up Method calling Pull Up Instance Variable.

3.3. Lessons on transformation and preconditions

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 preconditions while transformations would not need them. There would be a clear dichotomy in refactorings between their preconditions on one side and their transformations on the other side, both parts being independent and mutually exclusive.

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

- Transformations can have preconditions mainly for to check their applicability;
- Among the refactorings, the best candidates to be composed out of transformations are the ones from the precondition families *none* and *applicability check*;
- From an implementation point of view, we see that the preconditions 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.

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 *Elementary 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 (are composed of) more simpler refactorings. Finally, we identified some missed reuse opportunities: Refactorings that could be calling simpler refactorings but instead change the model directly.

4.1. Elementary Refactorings and Operations

As explained in the implementation overview, 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 in order 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.

In Table 1, we extracted the methods in RBEntity subclasses that perform code changes (elementary operations). There are four main subclasses: RBAbstractClass, RBClass, RBMethod, RBNamespace and 53 of these methods in total. The only operation from this list that is not currently being used by a refactoring is RBNamespace»renameClassVariable:to:in:around:. This is because any refactorings wishing to rename a class variable already does it by calling

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:

Table 1
RBEntity Model Operations called by Refactorings

RBClass»renameClassVariable:to:in:around: instead. There is code duplication here that should be removed.

From these operations, the refactorings are build. Table 2 presents the refactorings that are not referencing any other refactorings: *Elementary refactoring*, directly composed from the *Elementary operations*. The table also shows whether these Elementary refactorings are reused by other refactorings (*i.e., composite refactorings* discussed in Section 4.2). Finally, the table identifies those acting on a single entity of the source code, those marked with an asterisk (*). These last refactorings correspond to what Santos *et al.*, [14] defined as “Level one operators”:

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

Elementary refactorings not using other refactorings and how they might be reused by others – * means that the refactoring adds or remove only one entity in the model.

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 elementary refactoring of Table 2 and will discuss one case here: ADD CLASS refactoring. Listing 8 gives a part of its implementation. It exhibits an opportunity to re-implement the refactoring using a, simpler, transformation.

We see in the last line of the code that the refactoring can actually support the insertion of a class within a hierarchy. If the proposed parent of the new class has no sub-classes then the new class is created and nothing else happens. But this refactoring can accept subclasses of the parent class as parameters in order 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.

```
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 8: AddClassRefactoring

Therefore, this refactoring could be called Insert Class and could invoke a transformation Add Class that would only perform a class addition.

4.2. Explicit Composite Refactorings

To understand how to compose refactorings we analyzed *composite refactorings* that explicitly refer to other refactorings in their implementation. Table 3 presents all the composite refactoring we found. It is, in a sense, the counter part of Table 2 (showing elementary 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 9. Before recompiling the target method in the superclass (last statement), another refactoring has to be executed: Expand Referenced Pools.

```
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 9: RBPullUpMethodRefactoring implementation details: pullUp: method is called for each target method in the refactoring.

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

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

Table 3

Explicit composite refactorings and the refactorings they are composed of.

Since both Pull Up Method and Push Down Method are removing methods from one class and adding them to another, they could also use the Add and Remove refactorings in their transformations.

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. We list the implicit composite refactorings that we identified in Table 4, and Table 5 proposes some potential reuses in these implicit composite refactorings.

For example, Pull Up Instance Variable does not use Add Instance Variable (see Listing 10), and its preconditions do not take into account the preconditions of Add Instance Variable (which is checking if a name is valid, and if a variable with the same name does not already exist

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

Table 4
Implicitly composite refactorings: composite refactorings not reusing existing elementary ones.

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 means, however, that if a script using this refactoring is created, it could break the system just by using PULL UP INSTANCE VARIABLE with inadequate names. 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.

```
RBPullUpInstanceVariableRefactoring >> transform
  class allSubclasses do:
      [:each |
      (each directlyDefinesInstanceVariable: variableName)
        ifTrue: [each removeInstanceVariable: variableName]].
  class addInstanceVariable: variableName

RBPullUpInstanceVariableRefactoring >> preconditions
  ^Condition 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]
```

Listing 10: PullUpInstanceVariableRefactoring

```
RBPushDownInstanceVariableRefactoring >>transform
  class removeInstanceVariable: variableName.
```

Refactoring	could use…	…instead of
ChangeMethodName	RemoveMethod	removeMethod:
CreateAccessorsForVariable	AddMethod	compile:
CreateAccessorsWithLazyInitializationForVariable	AddMethod	compile:
DeprecateClass	AddMethod	compile:
GenerateEqualHash	AddMethod	compile:
GeneratePrintString	AddMethod	compile:
MoveInstVarToClass	AddMethod RemoveInstanceVariable	addMethod: and compile: removeInstanceVariable
MoveMethodToClass	AddMethod RemoveMethod	addMethod: and compile: removeMethod:
PullUpClassVariable	RemoveClassVariable	
PullUpInstanceVariable	RemoveInstanceVariable	
PushDownClassVariable	RemoveClassVariable	
PushDownInstanceVariable	RemoveInstanceVariable	
RemoveMethod	RemoveMethod	
SwapMethod	AddMethod and RemoveMethod	compile: and removeMethod:
TemporaryToInstanceVariable	RemoveInstanceVariable	

Table 5
Potential reuse in implicit composite refactorings

```
class subclasses do: [:each | each addInstanceVariable: variableName]
```

Listing 11: PullUpInstanceVariableRefactoring

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 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

~~We discussed~~

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 if could use Add Method, since its last statement does the same thing. Here we propose a AddMethodTransformation transformation (12) and a modification of RBPushDownMethodRefactoring to use this transformation (Listing 13).

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

Listing 12: Proposed RBAddMethodTransformation

Push Down Method could be altered in the following way and mantain its behavior.

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

Listing 13: 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 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 Replace Method and the 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 [5]

1. Check that the 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 the call sites of the oldName method and rewrite them to invoke the newName
4. Remove the old method 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.

As such, Replace Method cannot guarantee behavior preservation, it is not an actual refactoring. Listing 2 gives the #transform method of this refactoring. It invokes renameMessageSends on either a single class or all classes in the model, depending on an optional parameter (see Listing 14).

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

Listing 14: Rename Message Sends

RBChangeMethodNameRefactoring»renameMessageSends is used in both Rename Method as one of three operations, and on its own in Replace Method. It could be converted into a transformation (RBRenameMEssageSendTransformation) that would be more indicative of its behavior agnostic nature.

5.3. Discussion

The analyses performed and reported in previous sections show tracks for improving the reuse of logic between transformations and refactorings. We sketch here some general guidelines that future work will have to validate.

- Refactorings having only validity checking precondition are good candidate to transformations. They do not implement behavior preservation and as such can be used as transformation.
- Low-level API used by implicit refactorings are also opportunity to call their corresponding transformations.
- For the behavior preserving preconditions, it is unclear that it makes sense to turn them into mere transformations that could then been called by refactorings performing adequate behavior preserving analysis.

6. Related work

There are really few 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 have focus to discuss the reuse of transformation logic, they are the only works beside the PhD of D. Roberts formalizing refactoring implementation. Tichelaar [17, 18] presents some language independent refactorings on top of the FAMIX meta model [19] while Mayer *et al.*, presents a meta model to support cross language refactorings [20]. 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 are some works on refactoring engines for languages such as Erlang with tidier [21, 22], Wrangler [23], or refactoring for Ruby (RubyMine from jetbrains). But there is no explanation or information about the actual implementations of the refactoring engines themselves. Refactorings are simply explained from a user perspective.

Semantics-driven. Kesseli in his PhD [24] explores semantics-driven refactorings by opposition to syntactic refactorings (the ones mentioned 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 work focus on identifying the application of refactoring (Extract method application [25]), general refactorings [26]) with tools such as RefactoringMiner2.0. Another tool was developed to identify and refactor duplicate code in Java Projects [**?**]. Other works mined missed opportunities to refactor code (move method [27], missed polymorphism [28]. 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 support the understanding whether it makes sense to compose refactorings out of simple code transformations or other refactorings. For this we did a deep analysis of the current implementation of the original version of refactorings as defined by D. Roberts and J. Brant. We presented a classification of preconditions that identified four families of preconditions. In particular we learned that some preconditions are mainly checking the applicability of the refactorings and that as such the corresponding transformations can benefit from preconditions. We studied elementary refactorings (the ones that do not reuse any other refactorings) as well as the elementary operations offered by the system to actually implement the elementary refactorings. We wanted to understand whether the fact that not all the API of the system elementary operations are exposed as elementary transformation forced developers to implicit compose refactorings instead of simply reuse an elementary one. We studies explicit composite refactorings to understand how the preconditions and the elementary refactorings where interplaying. This led to the analysis of implicit composited refactorings (refactorings compose of multiple transformations but not using refactorings to perform them). We show that some implicit composite refactorings can easily be turned into explicit ones. Our analysis is the first step in the design of a co-existing and collaborating transformations and refactorings. The next step is to actually change implicit composite refactorings into explicit ones.

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

[6] D. Dig, C. Comertoglu, D. Marinov, R. Johnson, Automated detection of refactorings in evolving components, in: ECOOP, 2006, pp. 404–428.

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

[8] 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.

[9] 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.

[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] 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.

[12] E. Murphy-Hill, C. Parnin, A. P. Black, How we refactor, and how we know it, in: International Conference on Software Engineering (ICSE), 2009, pp. 287–297.

[13] 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.

[14] 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.

[15] 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.

[16] D. B. Roberts, Practical Analysis for Refactoring, Ph.D. thesis, University of Illinois, 1999.

[17] 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.

[18] S. Tichelaar, Modeling Object-Oriented Software for Reverse Engineering and Refactoring, Ph.D. thesis, University of Bern, 2001. URL: http://scg.unibe.ch/archive/phd/tichelaar-phd.pdf.

[19] 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, Technical Report, RMod – INRIA Lille-Nord Europe, 2011.

[20] 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.

[21] 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. URL: https://doi.org/10.1145/1599410.1599414. doi:10.1145/1599410.1599414.

[22] 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. URL: https://doi.org/10.1145/1596600.1596602. doi:10.1145/1596600.1596602.

[23] 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. URL: https://doi.org/10.1145/1411273.1411283. doi:10.1145/1411273.1411283.

[24] P. Kesseli, Semantic Refactorings, Ph.D. thesis, University of Oxford, 2018.

[25] 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 (2012) 2241–2260. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121212001057. doi:10.1016/j.jss.2012.04.013.

[26] 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.

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

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

A. Original list of refactorings

Original list of refactorings as in [16] 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 precondition families

We found four precondition 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 precondition.

C.1. Refactorings with precondition 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 precondition 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 precondition 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 precondition 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 preconditions 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: