Modular and Extensible Extract Method

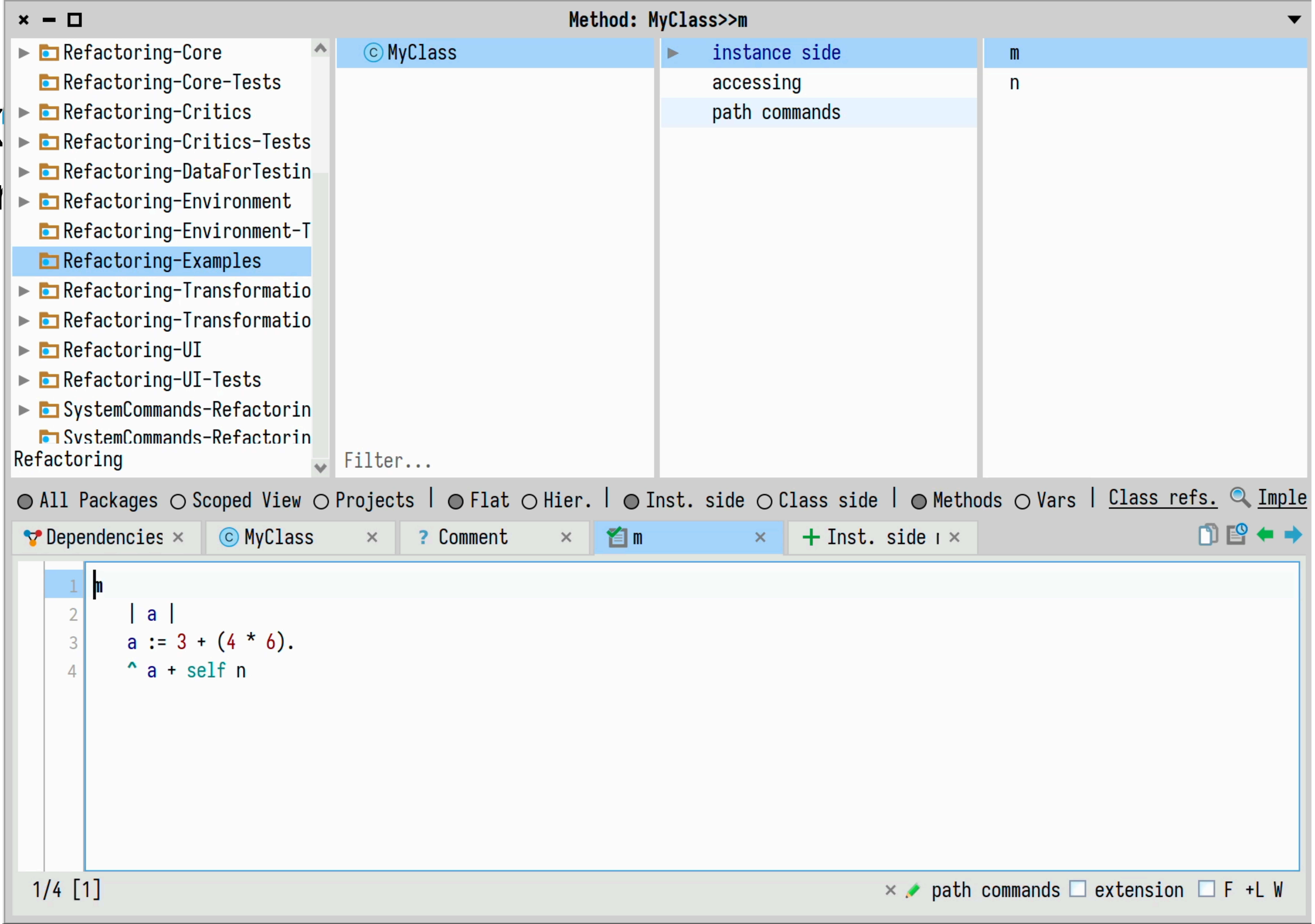
Balša Šarenac, Stéphane Ducasse, Guillermo Polito and Gordana Rakić

Introduction

- Refactoring
- Transformations
- Composition
- Extract Method
  - Source method
  - Extracted method

Ex

Extr



```
m
    | a |
    a := 3 + (4 * 6).
    ^ a + self n
```

Goals

- Enable users to define their own refactorings
- Redesign existing refactorings into modular definitions

Contributions

- Analysis of the existing Extract Method refactoring monolithic implementation.
- Definition of simplified rules for supporting the refactoring in the presence of multiple assignments, returns, and non-local returns.
- Definition of a modular Extract Method refactoring based on elementary operations.
- Reuse and extension of the Extract Method refactoring modular logic to define *domain-specific* refactorings: namely Extract SetUp refactoring for SUnit (Pharo's testing framework) and Extract with Pragma refactoring for Slang (virtual machine generator).

Legacy implementation

Pros and cons of legacy implementation

- Pros:
  - Mostly correct implementation
  - Correct precondition logic
- Cons:
  - Mixed calculations, precondition checking and transformation setup logic
  - Mixed transformation logic and user interaction
  - Monolithic implementation

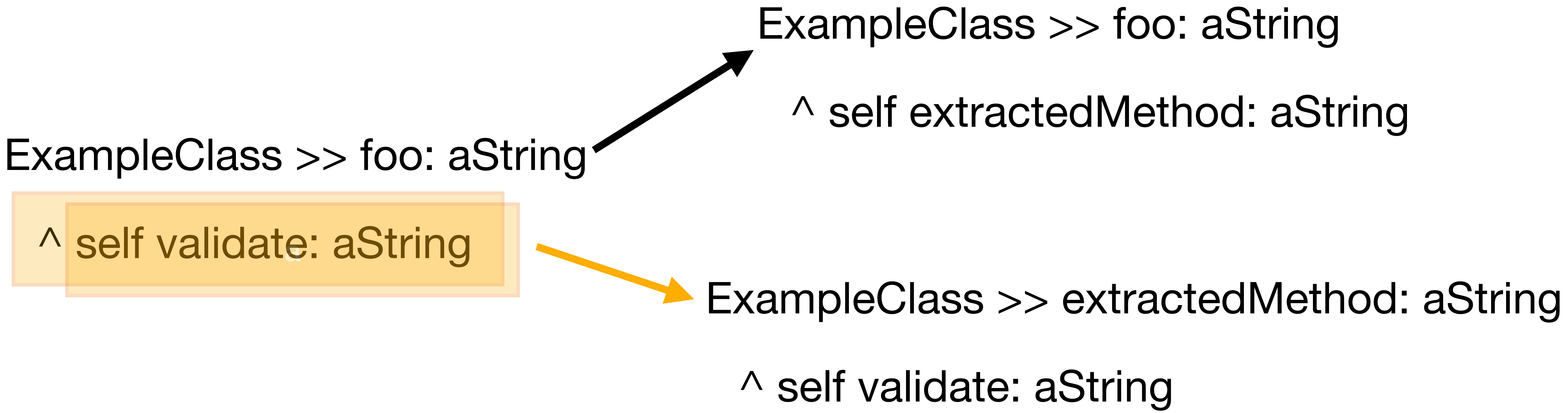
Analysis

Returns

- Returning a value from the extracted method
- Wrapping the extracted method’s invocation in the source method in a return statement

Analysis

Returns



```
ExampleClass >> foo: aString

  ^ self validate: aString
```

```
ExampleClass >> foo: aString

  ^ self extractedMethod: aString
```

```
ExampleClass >> extractedMethod: aString

  ^ self validate: aString
```

Analysis

Assignments

```
ExampleClass >> foo
  | a |
  a := 3 + (2 sqrt - 4) - (4 + 2 sqrt).
  ^ self validate: a
```

```
ExampleClass >> foo
  | a |
   a := self extractedMethod.
   ^ self validate: a
```

```
ExampleClass >> extractedMethod
  | a |
  a := 3 + (2 sqrt - 4) - (4 + 2 sqrt)
  ^ a
```

Analysis

Multiple assignments

```
ExampleClass >> foo
  | a b c d |
  a := 3.
  b := self bar: a.
  c := self baz: b.
  d := self doSomething.
   ^ self validate: c and: d
```

```
ExampleClass >> foo
  | b c d |
  b := self extractedMethod.
  c := self baz: b.
  d := self doSomething.
  ^ self validate: c and: d
```

```
ExampleClass >> extractedMethod
  | a b |
  a := 3.
  b := self bar: a.
  ^ b
```

Simplifications

Returns & Multiple assignments

- The extracted method can always return
- Add a return in the sender (source method) only if the last statement of the selection is the return statement
- Multiple assignments can be extracted only if at most one of the variables is used after the selection in the source method

Analysis

Non-local returning blocks

```
ExampleClass >> foo

  | c |

  c ifOdd: [ ^ true ].

  ^ self validate: c
```

Simplification

Non-local returning blocks

- Has single exit point

New architecture

- Prepare for execution
  - Calculate temporaries
  - Determine which assignment variables are used after the selection
  - Identify arguments for the extracted method
  - Determine if the source method needs to return the extracted method
- Precondition checking
  - Parse tree and selected code can be parsed
  - Selection is valid and extractable
  - Temporaries or assignments shouldn’t be read before written
  - Subtree has at maximum one assignment
  - Subtree has a single exit point
- Transformation
  - Create source for the extracted method
  - Search for method with equivalent parse tree
  - Create a message send to the extracted or found method
  - Perform transformations

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)

Journal of Computer Languages 80 (2024) 101273

Contents lists available at ScienceDirect

Journal of Computer Languages

journal homepage: www.elsevier.com/locate/cola

A new architecture reconciling refactorings and transformations

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

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

A R T I C L E   I N F O

Keywords:
Refactorings
Transformations
Preconditions
Source code transformation
Behavior preservation

A B S T R A C T

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

1. Introduction

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

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

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

Defining some specific code transformations such as ReplaceMessageSend and letting the Pharo developers define their own transformations are our long-term engineering goals. In this paper, we explore a new refactoring engine architecture to do so. Note that our goal is not

* Corresponding author.
E-mail addresses: balsasarenac@uns.ac.rs (B. Šarenac), nicolas.anquetil@inria.fr (N. Anquetil), stephane.ducasse@inria.fr (S. Ducasse), pablo.tesone@inria.fr (P. Tesone).

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

https://doi.org/10.1016/j.cola.2024.101273
Received 14 November 2023; Received in revised form 23 April 2024; Accepted 13 May 2024
Available online 31 May 2024

Results

```
ReCompositeExtractMethodRefactoring >> buildTransformationFor: newMethodName

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

Extract SetUp Method

```
ExampleTest >> testM
   | a |
   a := ComplexObject new.
   a doSomething
   self assert: a size equals: 4
ExampleTest >> testN
   | a |
   a := ComplexObject new.
   a doSomething.
   self assertEmpty: a
```

```
ExampleTest >> setUp
   super setUp.
   a := ComplexObject new.
   a doSomething
ExampleTest >> testM
   self assert: a size equals: 4
ExampleTest >> testN
   self assertEmpty: a
```

```
ReCompositeSetUpMethodRefactoring >> buildTransformationFor: newMethodName

   ^ OrderedCollection new
        add: (RBAddMethodTransformation
               model: self model
               sourceCode: newMethod newSource
               in: class
               withProtocol: (Protocol named: #running));
        add: (ReAddSuperSendAsFirstStatementTransformation
               model: self model
               methodTree: newMethod
               inClass: class);
        addAll: (assignments collect: [ :var | RBTemporaryToInstanceVariableRefactoring
               model: self model
               class: class
               selector: selector
               variable: var ]);
        add: (RBRemoveSubtreeTransformation
               model: self model
               remove: sourceCode
               fromMethod: selector
               inClass: class);
        add: (ReRemoveUnusedTemporaryVariableRefactoring
               model: self model
               inMethod: selector
               inClass: class name);
        yourself
```

Extract With Pragma

```
ExampleClass >> m
    <var: ‘c’ declareC: ‘int’>
    | c |
    c := 1 + 3.
    ^ c * self calculation
```

```
ExampleClass >> m
    <var: ‘c’ declareC: ‘int’>
    | c |
    c := self extractedMethod.
    ^ c * self calculation
```

```
ExampleClass >> extractedMethod
    <var: ‘c’ declareC: ‘int’>
    | c |
    c := 1 + 3.
    ^ c
```

```
ReCompositeExtractMethodWithPragmasRefactoring >> buildTransformationFor: newMethodName

    | messageSend |
    messageSend := self messageSendWith: newMethodName.
    ^ OrderedCollection new
        add: (RBAddMethodTransformation
                model: self model
                sourceCode: newMethod newSource
                in: class
                withProtocol: Protocol unclassified);
        add: (RBReplaceSubtreeTransformation
                model: self model
                replace: sourceCode
                to: messageSend
                inMethod: selector
                inClass: class);
        addAll: (pragmasToExtract collect: [ :p |
                  ReAddPragmaTransformation
                      model: self model
                      addPragma: p
                      inMethod: newMethod
                      inClass: class]);
        add: (ReRemoveUnusedTemporaryVariableRefactoring
                model: self model
                inMethod: selector
                inClass: class name);
        yourself
```

Conclusion

- Challenges when implementing the Extract Method in Pharo
- Leverage composition to create Extract Method
- Two use cases of domain specific refactorings:
  - Extract SetUp Method
  - Extract with Pragma