An Analysis of Inline Method Refactoring

Balša Šarenac, Stéphane Ducasse, Guillermo Polito and Gordana Rakić

Introduction

- Refactoring
- Composition
- Inline Method
  - Source method
  - Inline method

Goals

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

Inline method example

```
breakingChangePreconditions

^ { (RBCondition withBlock: [

    self findReceiverNode.

    true ]) }
```

```
breakingChangePreconditions

 ^ { (RBCondition withBlock: [

  | receiverNodes |

  receiverNodes := statementNodes collect: [ :each | each receiver ].

   receiverNodes asSet size = 1 ifFalse: [

      self refactoringError: 'All statements must have the same receiver' ].

   (receiverNodes first isLiteralNode or: [ receiverNodes first isVariable ]) ifFalse: [

     self refactoringWarning: 'The receiver is an expression. Proceed with caution' ].

  true ]) }
```

Contributions

- Analysis of the existing Inline Method refactoring monolithic implementation.
- Reuse and extension of the Inline Method refactoring modular logic to define *domain-specific* refactoring:
  - Inline Method 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

Canonicalization

- Transforming the inline method’s source code into a state that is inlinable.
  - Adding a return if it is not already explicitly defined,
  - Transforming guard clauses into if/else blocks (for example, ifTrue:ifFalse: in the case of Pharo),
  - Transform ifTrue: and ifFalse: into ifTrue:ifFalse:,
  - Removing non-local returns.

Analysis

Canonicalization

```
ExampleClass >> baz
    | c |
    c := self someMethod.
    c ifEmpty: [ ^ true ].
    ^ self validate: c
```

Before canonization

```
ExampleClass >> baz
    | c |
    c := self someMethod.
    ^ c
      ifEmpty: [ true ]
      ifNotEmpty: [ self validate: c ]
```

After canonization

Analysis

Cascades

- Right now handled using ifs
- Only supports inlining the last message from the cascade
- Makes the code very hard to change

New architecture

- Prepare for execution
- Precondition checking
- Transformation

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*

ARTICLE INFO

Keywords:
Refactorings
Transformations
Preconditions
Source code transformation
Behavior preservation

ABSTRACT

Refactorings are behavior-preserving code transformations. They are a recommended software development practice and are now a standard feature in modern IDEs. There are however many situations where developers need to perform mere *transformations* (non-behavior-preserving) or to mix refactorings and transformations. Little work exists on the analysis of transformations *implementation*, how refactorings could be composed of smaller, reusable, parts (simple transformations or other refactorings), and, conversely, how transformations could be *reused* in isolation or to compose new refactorings. In a previous article, we started to analyze the seminal implementation of refactorings as proposed in the 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

Current flow

- Inline Method refactoring:

Preprocessing

1. Check preconditions

Preconditions

2. Canonize the inline method,
3. Handle cascades and returns (+ more canonization),
4. Replace argument variables with values,
5. Substitute the message send with the inline method,
6. Remove dead code: empty and immediate blocks.

Transformations

Can we make this composite?

- Inline Method refactoring:

1. Canonize the inline method, Preprocessing
2. Check preconditions Preconditions
3. Handle cascades and returns Specialization
4. Rename conflicting temporaries
5. Replace argument variables with values,
6. Substitute the message send with the inline method,
7. Remove dead code, empty and immediate blocks. Transformations

Can we make this composite?

- Inline Method refactoring:

1. Canonize the method to be inlined, Preprocessing
2. Check preconditions Preconditions
3. Rename conflicting temporaries
4. Replace argument variables with values, Transformation - Preprocessing
5. Substitute the message send with the inline method, Transforming
6. Remove dead code, empty and immediate blocks. Cleanup

Transformations

Can we make this composite?

- Inline Method refactoring:

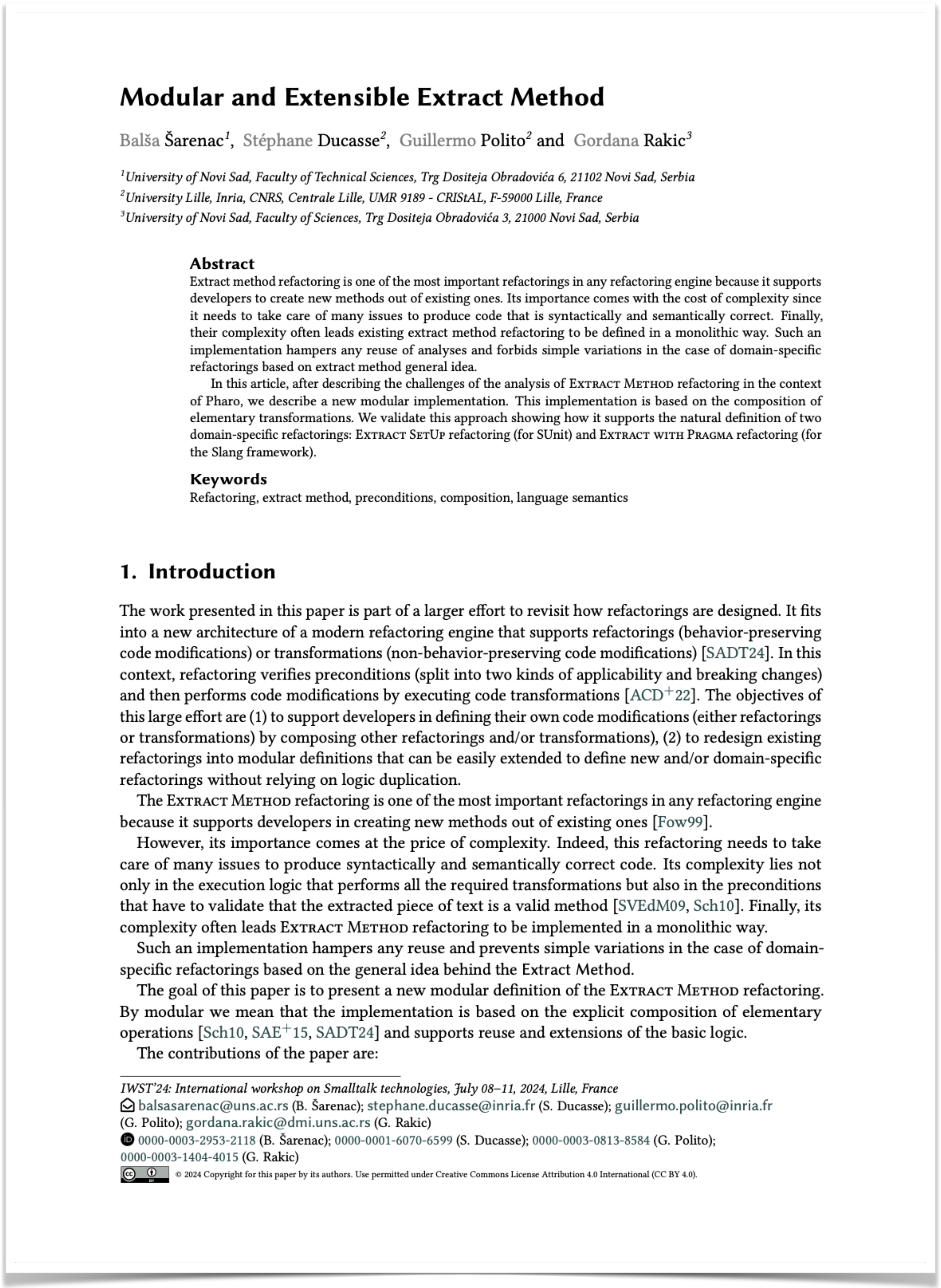
1. Canonize the method to be inlined (from transformation), Preprocessing

---

2. Check preconditions,
    1. Inlining overridden methods,
    2. Inlining bigger methods that will add statements before the target message.
    3. Inlining methods that send overridden super messages Preconditions

---

3. Inline method transformation. Transformations



Modular and Extensible Extract Method

Balša Šarenac[1], Stéphane Ducasse[2], Guillermo Polito[2] and Gordana Rakic[3]

[1]*University of Novi Sad, Faculty of Technical Sciences, Trg Dositeja Obradovića 6, 21102 Novi Sad, Serbia*
[2]*University Lille, Inria, CNRS, Centrale Lille, UMR 9189 - CRIStAL, F-59000 Lille, France*
[3]*University of Novi Sad, Faculty of Sciences, Trg Dositeja Obradovića 3, 21000 Novi Sad, Serbia*

Abstract
Extract method refactoring is one of the most important refactorings in any refactoring engine because it supports developers to create new methods out of existing ones. Its importance comes with the cost of complexity since it needs to take care of many issues to produce code that is syntactically and semantically correct. Finally, their complexity often leads existing extract method refactoring to be defined in a monolithic way. Such an implementation hampers any reuse of analyses and forbids simple variations in the case of domain-specific refactorings based on extract method general idea.

In this article, after describing the challenges of the analysis of Extract Method refactoring in the context of Pharo, we describe a new modular implementation. This implementation is based on the composition of elementary transformations. We validate this approach showing how it supports the natural definition of two domain-specific refactorings: Extract SetUp refactoring (for SUnit) and Extract with Pragma refactoring (for the Slang framework).

Keywords
Refactoring, extract method, preconditions, composition, language semantics

1. Introduction

The work presented in this paper is part of a larger effort to revisit how refactorings are designed. It fits into a new architecture of a modern refactoring engine that supports refactorings (behavior-preserving code modifications) or transformations (non-behavior-preserving code modifications) [SADT24]. In this context, refactoring verifies preconditions (split into two kinds of applicability and breaking changes) and then performs code modifications by executing code transformations [ACD$^+$22]. The objectives of this large effort are (1) to support developers in defining their own code modifications (either refactorings or transformations) by composing other refactorings and/or transformations), (2) to redesign existing refactorings into modular definitions that can be easily extended to define new and/or domain-specific refactorings without relying on logic duplication.

The Extract Method refactoring is one of the most important refactorings in any refactoring engine because it supports developers in creating new methods out of existing ones [Fow99].

However, its importance comes at the price of complexity. Indeed, this refactoring needs to take care of many issues to produce syntactically and semantically correct code. Its complexity lies not only in the execution logic that performs all the required transformations but also in the preconditions that have to validate that the extracted piece of text is a valid method [SVEdM09, Sch10]. Finally, its complexity often leads Extract Method refactoring to be implemented in a monolithic way.

Such an implementation hampers any reuse and prevents simple variations in the case of domain-specific refactorings based on the general idea behind the Extract Method.

The goal of this paper is to present a new modular definition of the Extract Method refactoring. By modular we mean that the implementation is based on the explicit composition of elementary operations [Sch10, SAE$^+$15, SADT24] and supports reuse and extensions of the basic logic.

The contributions of the paper are:

IWST'24: International workshop on Smalltalk technologies, July 08–11, 2024, Lille, France
balsasarenac@uns.ac.rs (B. Šarenac); stephane.ducasse@inria.fr (S. Ducasse); guillermo.polito@inria.fr (G. Polito); gordana.rakic@dmi.uns.ac.rs (G. Rakic)
0000-0003-2953-2118 (B. Šarenac); 0000-0001-6070-6599 (S. Ducasse); 0000-0003-0813-8584 (G. Polito); 0000-0003-1404-4015 (G. Rakic)

Comparison to Extract Method?

```
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
```

What do we do with method with Pragma

```
ExampleClass >> m
   | a |
   a := self exampleMethod.
   ^ a * self calculation
```

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

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

Inline with pragma composite

- Inline Method with pragma refactoring:

1. Canonize the method to be inlined,

Preprocessing

2. Check preconditions,

Preconditions

3. **Handle pragmas,**
4. Inline method transformation.

Transformations

Conclusion

- Challenges when implementing the Inline Method in Pharo
- Leverage Decorator and Specialization to simplify Inline Method
- Inline Method with Pragma domain specific refactoring