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) [1]. In this context, refactoring verifies preconditions (split into two kinds of applicability and breaking changes) and then performs code modifications by executing code transformations [2]. 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 [3].

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 [4, 5]. 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 [1, 5, 6] and supports reuse and extensions of the basic logic.

The contributions of the paper are:

IWST 2024: International Workshop on Smalltalk Technologies, July 9–11, 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)

- 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 definition of 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 [7] (Pharo's testing framework) and Extract with Pragma refactoring for Slang [8] (virtual machine generator).

In Section 2, the paper starts with the analysis of the legacy Extract Method refactoring implementation as available in Pharo 12: it shows that (1) there is a long list of precondition checks for assignments, returns, and temporaries, (2) a lot of precondition checks are mixed with setup and execution logic, (3) the execution logic can only happen because some preconditions had strong side effects, and (4) the execution is mixed with user interaction concerns. Section 3 presents the challenges for the definition of a viable and simple set of constraints that can ease both precondition checking and execution logic. Section 4 describes a new modular implementation of the Extract Method refactoring method. Finally, we validate our approach showing how the definition of domain-specific refactorings can reuse the logic of our new Extract Method refactoring modular implementation.

2. Analysis of the legacy implementation

In this section, we study the legacy (monolithic) implementation of the Extract Method refactoring. First, we summarize the essence of the refactoring then we show a detailed cursive description of each of the steps of this refactoring and finally we discuss the pros and cons. We only focus on the pros and cons of the Extract Method refactoring. The interested reader should refer to [1, 9] for a more general architectural analysis.

2.1. Logic and process

The logic of the Extract Method refactoring is to allow the user to specify a portion of source code from a method to be moved to a new separate method and replace the old code with a call to the new method. The Extract Method refactoring is often used to create a hook method within a template method, as seen in the Hook and Template Design pattern [10].

Tools may check if there is already a method with the same body in the class hierarchy. In such cases, instead of creating a new method, the existing method is called.

The key points that the refactoring should consider are:

- The selected code must be a valid sequence of instructions.
- The modified source method should be valid.
- The execution path of the modified method should be the same as its original version.

2.2. Detailed analysis

In this section, we present in a cursive manner, the logic of the existing Extract Method implementation as defined in Pharo 12, and as has been inherited from the original code definition [11, 12], named such version the legacy version in the rest of the paper. The Ph.D. thesis of Roberts [13] does not include a discussion of such refactoring probably because of its complexity. Presenting the list of steps is interesting because it illustrates the inherent complexity of the legacy monolithic implementation. We then present a short analysis of the situation.

Precondition checking. After the EXTRACT METHOD refactoring is invoked it starts by checking preconditions. There are multiple preconditions to check and in the legacy implementation, the precondition checking logic also contains setup for execution as well. Here are the precondition steps performed by the legacy implementation:

- Initial preconditions include checking that the method from which we are extracting code (we will call this method "source method" in the rest of the paper) exists in the given class.
- Next, it checks that the code selected for extraction is parsable and valid Pharo code (syntactically and semantically correct). As a result, it produces an AST node.
- After that, it creates a new RBMethodNode from that node. This node serves as the basis for building the extracted method.
- Then, the refactoring creates a parse tree for the source method and fails if it is unable to do so. This tree is used to perform various checks and transformations of the source method.
- The refactoring searches the parse tree of the source method to identify the subtree that represents the code selected for extraction.
- Next, the refactoring replaces the selected code with a placeholder value. The placeholder value is just a symbol that will later be replaced with a call to the extracted method.
- Additionally, if the last statement of the extracted code is an assignment, the refactoring asks the user whether to extract that assignment as well.
- If the user does not want to extract the assignment the placeholder is changed to an assignment expression, where the extracted method is the value to be assigned to the variable of the last assignment. This means that the extracted method should return the value to be assigned in the last assignment, and the refactoring should preserve the behavior by assigning that method back to its variable.
- After setting up the placeholder the refactoring checks for some special cases that would produce invalid code: (1) whether the selected code is on the left-hand side of an assignment, and (2) whether the selected code is the first of cascaded messages.
- After handling the special cases, the refactoring checks if it needs to add a return in the extracted method. It needs to add a return message if the selected code should be used as a return value. For example, if the developer extracts the right-hand side of an assignment, the extracted method should return the resulting value to preserve the behavior. More on this later in Section 3.1.
- Next, the refactoring checks if the selection contains non-local returning blocks whose extraction would impact behavior. More on this later in Section 3.2.
- After that, the refactoring checks and wraps, if necessary, a return statement around the placeholder symbol in the source method. This is required when the extracted code contains the source method's return value.
- After dealing with returns, the refactoring deals with temporaries. First, it calculates the new set of temporaries in the source method.
- Then, it calculates which temporaries are accessed in the extracted method and which are assigned in the extracted method. Out of the assigned temps, it checks if there is more than one of them that is being referenced in the new source method (the one with a placeholder instead of the code for extraction). If it exists, an error is raised. If there is only a single reference, it adds a return node that will return that variable from the extracted method.
- Next, it removes referenced variables from the assigned variables and checks if the resulting set of variables is read before written in the extracted method.
- Finally the refactoring removes unnecessary temps from the source method and adds the required temps to the extracted method.

This concludes the precondition checking as well as the preparation for execution for the EXTRACT METHOD refactoring.

Code transformation. The next stage is the actual code transformation stage:

- The first step in this stage is to find a name for the extracted method. It checks with the user if it should search for similar methods in the hierarchy instead of creating a new method.
- If the user wants to search and if it finds a similar method, it sets it as the new name and properly configures parameters.
- If the user does not want to search for a similar method, it asks the user to give a name for the extracted method.
- Once the name is configured, what is left is to replace the placeholder in the source method with the new name and correctly fill in the arguments.
- Finally, it changes the extracted method's selector.

2.3. Discussion

Positive points.

- *Mostly correct implementation.* Extract method being the core refactoring and one of the most used ones, it has to be bulletproof. Pharo 12's implementation of this refactoring can be considered solid since it only had three active issues on the issue tracker before we started working with it. During the mutation testing of the Extract Method refactoring we have found three additional issues that we reported to the issue tracker, and a couple more that were not reported yet. The number of issues might not be the perfect measure of quality, but it can be considered a decent one.
- *Correct precondition logic.* We have performed an extensive analysis of the preconditions of the Extract Method refactoring and it is implemented correctly. However, the implementation of these preconditions is fairly complex and hard to reason about.

Negative points. Maintaining this refactoring over the years has introduced many patches. These patches increased maintenance costs and increased the overall complexity of the code. As a result, making modifications to the Extract Method refactoring is tedious, and extending it is hard. For example, there is an implementation of Extract SetUp refactoring, which is based on Extract Method refactoring, and it does not work except in some happy paths. Attempts to fix it were unsuccessful due to the complexity of the Extract Method refactoring itself.

- *Mixed calculations, precondition checking, transformation setup logic.* To perform the extraction, the refactoring needs to check preconditions. In the case of Extract Method refactoring, it needs to perform some calculations (i.e. which arguments should be passed to the extracted method). Calculating that information is mixed with precondition checking and that is making the whole code more complex.
- *Mixed transformation logic and user interaction.* In the transformation phase, the refactoring asks the user whether to search for an existing method with an equivalent parse tree that can be used instead of creating a new method. This prevents duplication of logic when the code developer is extracting code that already exists. Also, in the transformation phase, the developer is prompted to give a name to the extracted method. This mixing of transformation logic and user interaction is not considered a good practice and makes the Extract Method refactoring deal with multiple responsibilities [1].
- *Monolithic implementation.* Anquetil et al. [2] pointed out that Pharo has an extensive set of transformations. Those transformations are wrappers around a program model that ensure only valid transformations are executed. Extract Method refactoring is performing all transformations on its own, and thus we have repeated code and a missed reuse opportunity.

3. Points of concern and logic simplification

In this section, we present points to address when extracting methods namely returns and multiple assignments. Then we present some simplifications to ease the refactoring checking.

3.1. Returns and multiple assignments

We discuss the two concepts of returning in Extract Method refactoring: (1) when to return a value from the extracted method, and (2) when to return the call to the extracted method in the source method.

3.1.1. Extracting return statement when to return a value from the extracted method

It is important to add a return statement to the newly extracted method when necessary. Pharo's refactoring engine relies on AST nodes to determine if the extracted code needs to return something. The AST logic for determining if a node is needed as a return value is quite complex and varies between nodes. We will not go into details but explain the point via an example: when extracting the right-hand side of an assignment, the extracted code needs to return that value so that the result of the newly extracted method call is correctly assigned to the variable.

Let's look at the example in Listing 1.

```
ExampleClass >> foo: aString
    ^ self validate: aString
```

Listing 1: Example method that returns a result of a message send

We should be able to extract self validate: aString, and the resulting code should add a return in the extracted method, since the same return value needs to be returned from the source method (Listing 2).

```
ExampleClass >> foo: aString
    ^ self extractedMethod: aString
ExampleClass >> extractedMethod: aString
    ^ self validate: aString
```

Listing 2: Result of extract method refactoring when performing it on the assignment from Listing 1

Multiple assignments. Pharo supports returning a single value from a method. When we extract an assignment, we can return the value of that assignment from the extracted method. However, we do not always need to return that value; it is only necessary when the assignment's variable is used later in the source method (after the extracted selection). Listing 3 illustrates this when the user extracts 3 + (2 sqrt - 4) - (4 + 2 sqrt). The result is displayed in Listing 4.

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

Listing 3: Example method with a single assignment

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

Listing 4: Result of extract method refactoring when performing it on the assignment from Listing 3

Based on this, we can support extracting more than one assignment, if one, and only one is used after the selection that is being extracted. In that case, we can extract the method and return only the variable that is being used after the selection. Let's examine an example in Listing 5.

```
 1  ExampleClass >> foo
 2      | a b c d |
 3      a := 3.
 4      b := self bar: a.
 5      c := self baz: b.
 6      d := self doSomething.
 7      ^ self validate: c and: b
```

Listing 5: Example method with multiple assignments

If the extracted code contains lines from 3 to 4, the expected result is shown in Listing 6. That selection contains two assignments, but only one (b) is used later in the code.

```
 1  ExampleClass >> foo
 2      | b c d |
 3      b := self extractedMethod.
 4      c := self baz: b.
 5      d := self doSomething.
 6      ^ self validate: c and: b
 7
 8  ExampleClass >> extractedMethod
 9      | a b |
10      a := 3.
11      b := self bar: a.
12      ^ b
```

Listing 6: Result of extract method refactoring when performing it on the assignments on lines 3 and 4 from Listing 5

We can also extract line 6 since variable d is not used on lines after 6 (which is only line 7). We should not be able to extract lines 4 and 5, since they contain two assignments whose variables are used after the extracted section (line 7 uses c and b).

3.1.2. Simplifications

We propose two simplifications of the logic. Note that our goal is to get simple rules that can ease the analysis without making too many unnecessary changes to the original code. One for checking when to add a return in the extracted method, and one for checking when to wrap the resulting extracted method invocation in a return statement.

The extracted method can always return. To simplify the logic of calculating whether a return is required to be added to the extracted method, we will always add a return statement to the extracted method. We should always be able to return the last statement from the extracted method without impacting behavior. However, we still need to calculate which value to return: the one that is used in the source method or the last statement (which is the default case).

When to add a return in the sender (source method). We can simplify the logic of adding a return statement in the source method that returns the extracted method's return value. This should only be the case if the selection to be extracted has a return statement as the last statement in the selection.

3.2. Non-local returning blocks

Pharo has blocks which are lexical closures. Blocks in the presence of return statements behave like an escaping mechanism. A block with an explicit return statement is called a non-local returning block.

The evaluation of the block returns to the block home context sender (i.e., the context that invoked the method creating the block) [14].

3.2.1. Non-local concerns for Extract Method refactoring

While non-local returning blocks are useful for guard statements, in Pharo it is discouraged to use them for other cases. Non-local returning block change method execution flow, and when extracting non-local returning blocks, we can change the source method's execution flow. Not all extractions of non-local returning blocks are unsafe.

If one of the statements contains a non-local returning block, it is possible to safely extract it only if the execution paths of the source method are preserved after the extraction. This basically means that the extracted code containing non-local returning blocks should also contain the method's return statement. In terms of selection, this means that all lines of the method starting from the non-local returning block to the end of the method should be included. The selected code should either always return something or flow from the beginning to the end without non-local returning blocks.

For example, we can extract lines 3 to 5 from Listing 7, since when extracted, the new method will include all returns of that method and preserve all execution paths. It should not be possible to extract only line 4 or lines 3 and 4 since they contain non-local returning blocks and there is another return after the selection.

```
1  ExampleClass >> foo
2      | c |
3      c := self extractedMethod.
4      c ifOdd: [ ^ true ].
5      ^ self validate: c
```

Listing 7: Example method that contains non-local returning block

3.2.2. Simplifications

Has single exit. We can simplify the precondition that checks if a selection containing a non-local returning block can be extracted. If the last statement of the selection is a return statement, we can safely extract it since all returns are selected after the non-local returning block. If the last statement is not a return statement, then there should not be a non-local returning block in any of the statements in the selection.

4. Extract Method refactoring as a sequence of elementary operations

The modular implementation of the Extract Method refactoring is based on a new refactoring architecture [1] for refactorings and transformations. This architecture is based on the explicit composition of elementary operations (transformations or refactorings).

The new architecture for refactorings described in [1], supports the refactoring logic into a kind of general template method structured around several main steps. In this case, the refactoring consists of three main steps: performing various computations to prepare for the execution, checking preconditions, and finally performing the code modification.

- **Preparation for precondition checking and execution.** In the first step, the Extract Method refactoring parses the method and the code selected for extraction. If either of them cannot be parsed, the remaining computations are skipped, and the refactoring is aborted. The remaining computations include: calculating temporaries, determining which assignments are used by which variables, and identifying arguments for the extracted method, and if the source method needs to return the extracted method (*i.e.,* wrap it in a return statement).

- **Precondition checking.** The refactoring checks whether the parse tree of the source method and the extraction subtree were parsed correctly. It then verifies if the selection is valid and can be extracted. The refactoring checks for:
  – temporaries or assignments that are read before being written,
  – that the subtree has at maximum one assignment, and
  – that the subtree has a single return point.
- **Transformation phase.** First, a new method node is created based on the selected subtree, temporaries, assignments, and arguments. Next, it tries to find an existing selector that has an equivalent tree to the selected subtree (*i.e.,* the created method node). If a match is found, a new method is not created, instead the found selector is invoked with the equivalent tree. If an equivalent method is not found, the extraction is performed in three steps (see Listing 8):
  – The method node is compiled with ADD METHOD transformation.
  – The selected code in the source method is replaced with an invocation to the extracted method.
  – Unused temporaries are removed from the source method.

Listing 8 is the code that performs the transformation phase of the refactoring.

```
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
```

Listing 8: buildTransformationFor method of composite extract method refactoring

Listing 8 shows that the composite EXTRACT METHOD refactoring is the composition of elementary transformations (RBAddMethodTransformation, RBReplaceSubtreeTransformation) and refactorings (ReRemoveUnusedTemporaryVariableRefactoring).

5. Evaluation

In this section, we highlight the key differences between two implementations, and showcase how composition enables extensibility.

5.1. Comparison of legacy and modular implementation

The modular implementation introduces multiple improvements over the legacy implementation. For example:

- **Separation of concerns.** The modular implementation leverages the new architecture and has a clear separation of concerns. There is no mixing of precondition checking logic with execution setup logic, or mixing of execution logic with user interaction.

- **Simplified preconditions.** Compared to the legacy implementation, the modular implementation has significantly simplified preconditions. This makes them easy to understand and reason about. We have compared the two implementations and have not found any cases where legacy and modular implementation's preconditions differ. Furthermore, we have run mutation tests for both implementations and the modular implementation has the same number of failing tests.
- **Descriptive definition of transformations.** The modular implementation takes a descriptive approach to defining which transformations should be applied. Instead of manually invoking the program model API, like the legacy implementation does, the modular implementation leverages existing transformations. We have already written about the benefits of this approach in [1].
- **Code metrics.** The modular implementation has around 5% fewer lines of code and fewer methods (33 compared to 41 in the legacy implementation). Furthermore, it has a 49% smaller cumulative cyclomatic complexity (94 compared to 41) and 37% smaller average cyclomatic complexity per method.

5.2. Composition at work

In this section, we show how the modular definition of the Extract Method refactoring eases the definition of new refactorings or transformations [2]. We illustrate this with two new refactorings the Extract SetUp refactoring and the Extract with Pragma refactoring.

- The Extract SetUp refactoring helps the developer define which part of test methods should be automatically turned into a setUp method — The setUp method in the SUnit and JUnit 3.0 frameworks is the method that creates the test fixture before any test method execution [7, 15].
- The Extract with Pragma refactoring helps the developer extract methods that are annotated with domain-specific annotations.

Defining the two new refactorings the Extract SetUp refactoring and the Extract with Pragma refactoring *without* our new composite Extract Method refactoring leads to several problems.

The legacy implementation of the Extract SetUp refactoring has the following limits:

- A large part of the precondition and transformation logic is duplicated but slightly modified.
- The logic is convoluted and difficult to follow - the actual implementation is only working on limited scenarios.

5.3. Composite Extract SetUp refactoring

The modular design described in the previous section allows for easy extension and creation of specialized refactorings based on it.

Required modifications include changes to preconditions: the precondition that only one assignment maximum is allowed was removed in favor of two new preconditions. The new preconditions check that the refactoring does not override an existing setUp method and that the refactoring is performed in a class that is a subclass of TestCase. Inheriting ReCompositeSetUpMethodRefactoring from ReCompositeExtractMethodRefactoring makes it easy to adapt the preconditions and reuse existing ones.

Besides precondition changes the transform step changes significantly:

- Add a new setUp method based on the selected subtree.
- Add a super send to the setUp method.
- Transform all assignment variables to instance variables.
- Remove the selected subtree from the source method.
- Remove unused temporaries from the source method.

The resulting buildTransformationFor method is shown in Listing 9.

```
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
```

Listing 9: buildTransformationFor method of composite extract setUp method refactoring

Thanks to the declarative and composite nature of the Extract Method refactoring making the required changes to create the setUp method refactoring is rather straightforward, and also easier to debug, maintain, and extend. The nature of the transformation logic is different going from a more imperative to a more declarative one.

5.4. Composite Extract with Pragma refactoring

The Pharo virtual machine is written in a subset of Pharo that can be transpiled to C using a VM generator called Slang [8, 16]. Slang utilizes Pharo to C transpilation: taking Slang code as input, which consists of Pharo code with method annotations [17] and certain constraints. For example, to be transpiled to C, the VM must not contain polymorphic method definitions, runtime object allocations (new), or exceptions. Non-local returns are transpilable only when blocks containing them do not get orphan of their outer context, which can occur when block closures are stored in instance variables for future use. By leveraging annotations with different semantics, Slang generates a C file, that once compiled, becomes the Pharo VM [18, 19].

In Slang [8], methods are annotated with compilation directives and type information. An example of this can be seen in Listing 10 which shows the annotation <var:declareC:>. Certain annotations provide important information such as whether the transpiled method should be inlined or not, should not be transpiled, its return type, or whether the method is an API, etc. Altering these annotations may cause issues during transpilation or with the resulting virtual machine. Therefore, it is crucial for developers to have the ability to decide whether a pragma should be extracted together with a portion of the method.

```
findFirstInString: aString  inSet: inclusionMap  startingAt: start

    | i stringSize |
    <primitive: ''primitiveFindFirstInString'' module: ''MiscPrimitivePlugin''>
    <var: #aString declareC: ''unsigned char *aString''>
    <var: #inclusionMap  declareC: ''char *inclusionMap''>

    inclusionMap size ~= 256 ifTrue: [ ^0 ].

    i := start.
    stringSize := aString size.
    [ i <= stringSize and: [ (inclusionMap at: (aString basicAt: i) + 1) = 0 ] ] whileTrue: [
        i := i + 1 ].

    i > stringSize ifTrue: [ ^0 ].
    ^i
```

Listing 10: Example method that contains Slang pragmas

Extending the Composite Extract method refactoring so that it includes the extraction of pragmas along with the code is just as straightforward and intuitive as introducing the Extract SetUp refactoring. The following steps are involved:

- Firstly, it is necessary to determine which pragmas need to be moved or copied. This involves adding another method to prepare for the execution hook. In this proof of concept, we have enabled support for the type pragma #type declareC .
- Secondly, the identified pragmas are added to the new method. To achieve this, an additional transformation has been included in the list of composite transformations (lines 18-23 in Listing 11). This simple modification allows for the extraction of pragmas.

We are currently working on generalizing of this refactoring so that it can support multiple pragmas. However, for the purposes of this proof of concept, we have focused on type pragmas in the Slang language, which is used to develop the Pharo VM.

```
1  ReCompositeExtractMethodWithPragmasRefactoring >> buildTransformationFor: newMethodName
2
3     | messageSend |
4     messageSend := self messageSendWith: newMethodName.
5     ^ OrderedCollection new
6          add: (RBAddMethodTransformation
7                 model: self model
8                 sourceCode: newMethod newSource
9                 in: class
10                withProtocol: Protocol unclassified);
11         add: (RBReplaceSubtreeTransformation
12                model: self model
13                replace: sourceCode
14                to: messageSend
15                inMethod: selector
16                inClass: class);
17         addAll: (pragmasToExtract collect: [ :p |
18                   ReAddPragmaTransformation
19                      model: self model
20                      addPragma: p
21                      inMethod: newMethod
22                      inClass: class]);
23         add: (ReRemoveUnusedTemporaryVariableRefactoring
24                model: self model
25                inMethod: selector
26                inClass: class name);
27         yourself
```

Listing 11: buildTransformationFor method of composite *with pragmas*

6. Related work

The bulk of the related work regarding the Extract Method refactoring is based on identifying places to perform the extract method or to automate this refactoring. This can be seen in a systematic literature review by AlOmar *et al.,* [20] where authors describe a review of the Extract Method body of work including 89 papers that mainly focus on papers that automate extract method refactoring and aid the developer when applying refactorings. There is not much said about the composite extract method or underlying implementation.

Regarding specifying or implementing the Extract Method refactoring, we have searched intensively the literature and found really limited material. One of the rare papers on this is by Schäffer *et al.,* [4, 5]. Besides Schäffer *et al.,* we have found a master thesis [21] that documents Java preconditions and extract method, with the underlying goal of automating Extract Method refactoring. One would assume that Fowler's book on refactoring [22] has some specifications, however, there is no mention of preconditions and the description is meant for developers who are manually performing the refactoring and relying on the build system to guide them through the process and fix any issues along the way.

Schäffer [5] in his Ph.D. thesis presents a way to compose the Extract Method refactoring out of 5 micro-refactorings. This approach is similar to what we want to achieve as our end goal. It presents a comprehensive analysis of problems that need to be addressed while developing Extract method refactoring for the Java language. These steps are not fully applicable to Pharo since Java and Pharo differ significantly. The main idea, however, is to create an anonymous function and then promote it to a regular method that can be used in Pharo. Compared to our approach Schäffer performs refactorings at each step and continually preserves behavior, whereas our approach performs transformations.

Horpácsi *et al.,* [23] has a similar approach to composition as Schäffer, but they expand on it and introduce semi-automatic formal verification of refactorings. The authors utilize refactoring schemes that are verified algorithmic skeletons whose instances can be automatically verified. They develop refactorings based on those schemes and therefore enable users to easily create verifiable refactorings. While this paper does not address the extract method, its approach is similar to ours in terms of composition. However, we do not have any formal proofs of our implementations. It is our future work to further investigate ideas from this paper and assess their applicability to Pharo.

Thy *et al.,* [24] introduced REM, an IntelliJ IDEA plugin for the Rust language. The plugin can perform extract methods refactoring that is guaranteed to produce well-typed Rust code. The authors analyzed the challenges of implementing Extract Method refactoring for Rust and extended the existing IntelliJ IDEA plugin for Rust refactoring by introducing extra transformations that repair and produce well-typed code. Compared to our approach it is similar where both papers present composite Extract Method refactoring, and discuss the challenges of implementing it. However, since the target languages are different the analysis focuses on specific language properties (*e.g.,* the authors of REM plugin focused on lifetimes and borrow checker features of Rust).

Murphy-Hill *et al.,* [25] present three tools to assist in refactoring and performed a user study evaluating those tools. Tools are focused on lowering the number of failed attempts when performing Extract Method refactoring. Two tools are designed to help with the selection range for the extract method, and the third tool is focused on giving better visual feedback for failed preconditions. The authors give usability recommendations for refactoring tools both for code selection and for displaying violated preconditions. Our paper focused more on the actual implementation of the Extract method refactoring, we plan to improve the user experience of our engine by relying on refactoring drivers [1].

7. Conclusion

In this article, we have described the challenges of implementing Extract Method refactoring in the Pharo programming language. We demonstrated how composition can be leveraged to make the Extract Method refactoring more modular and extensible. Additionally, we have described the key differences between legacy and modular implementation. Modular implementation brings various improvements such as better separation of concerns, simplified preconditions, a descriptive approach to defining transformations to be applied, and improvements across various software metrics. Finally, we have demonstrated the extensibility of this new modular Extract Method refactoring: Extract SetUp refactoring (for SUnit) and Extract with Pragma refactoring (for the Slang framework).

References

[1] B. Sarenac, N. Anquetil, S. Ducasse, P. Tesone, A new architecture reconciling refactorings and transformations, Journal of Computer Languages (2024) 101273. doi:https://doi.org/10.1016/j.cola.2024.101273.

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

[3] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Professional, 1999.

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

[5] M. Schäfer, Specification, Implementation and Verification of Refactorings, Ph.D. thesis, Oxford University Computing Laboratory, 2010.

[6] 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.

[7] S. Ducasse, G. Polito, J. P. Sandoval, Testing in Pharo, Book on Demand – Keepers of the lighthouse, 2003. URL: http://books.pharo.org.

[8] E. Miranda, C. Béra, E. G. Boix, D. Ingalls, Two decades of Smalltalk VM development: live VM development through simulation tools, in: Proceedings of International Workshop on Virtual Machines and Intermediate Languages (VMIL'18), ACM, 2018, pp. 57–66. doi:10.1145/3281287.3281295.

[9] N. Anquetil, J. Delplanque, S. Ducasse, O. Zaitsev, C. Furhman, Y.-G. Guéhéneuc, What do developers consider magic literals? a smalltalk perspective, Information and Software Technology (2022). doi:10.1016/j.infosof.2022.106942.

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

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

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

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

[14] S. Ducasse, C. Bera, Blocks: a detailed analysis, in: Deep Into Pharo, Square Bracket Associates, 2013, p. 25. URL: http://books.pharo.org.

[15] K. Beck, E. Gamma, Test infected: Programmers love writing tests, Java Report 3 (1998) 51–56. URL: http://members.pingnet.ch/gamma/junit.htm.

[16] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay, Back to the future: The story of Squeak, a practical Smalltalk written in itself, in: Proceedings of Object-Oriented Programming, Systems, Languages, and Applications conference (OOPSLA'97), ACM Press, 1997, pp. 318–326. doi:10.1145/263700.263754.

[17] S. Ducasse, E. Miranda, A. Plantec, Pragmas: Literal messages as powerful method annotations, in: International Workshop on Smalltalk Technologies IWST'16, Prague, Czech Republic, 2016. doi:10.1145/2991041.2991050.

[18] G. Polito, P. Tesone, S. Ducasse, L. Fabresse, T. Rogliano, P. Misse-Chanabier, C. H. Phillips, Cross-ISA Testing of the Pharo VM: Lessons Learned While Porting to ARMv8, in: Proceedings of the

18th international conference on Managed Programming Languages and Runtimes (MPLR ’21), Münster, Germany, 2021. URL: https://hal.inria.fr/hal-03332033. doi:10.1145/3475738.3480715.

[19] Q. Ducasse, G. Polito, P. Tesone, P. Cotret, L. Lagadec, Porting a jit compiler to risc-v: Challenges and opportunities, in: Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes (MPLR ’22), Brussels, Belgium, 2022. URL: https://hal.archives-ouvertes.fr/hal-03725841.

[20] E. A. AlOmar, M. W. Mkaouer, A. Ouni, Behind the intent of extract method refactoring: A systematic literature review, IEEE Transactions on Software Engineering 50 (2024) 668–694. doi:10.1109/TSE.2023.3345800.

[21] J. Hubert, Masterarbeit Implementation of an Automatic Extract Method Refactoring, Master’s thesis, University of Stuttgart, 2019. URL: https://api.semanticscholar.org/CorpusID:201111123.

[22] M. Fowler, Refactoring: Improving the Design of Existing Code, 2 edition ed., Addison-Wesley Professional, Boston, 2018.

[23] D. Horpácsi, J. Köszegi, Z. Horváth, Trustworthy refactoring via decomposition and schemes: A complex case study, in: VPT@ETAPS, 2017.

[24] S. Thy, A. Costea, K. Gopinathan, I. Sergey, Adventure of a lifetime: Extract method refactoring for rust, Proc. ACM Program. Lang. 7 (2023). URL: https://doi.org/10.1145/3622821. doi:10.1145/3622821.

[25] E. Murphy-Hill, A. P. Black, Breaking the barriers to successful refactoring: observations and tools for extract method, in: Proceedings of the 30th International Conference on Software Engineering, ICSE ’08, Association for Computing Machinery, New York, NY, USA, 2008, pp. 421–430. URL: https://doi.org/10.1145/1368088.1368146. doi:10.1145/1368088.1368146.