Building Blocks for Object-Oriented Refactoring Engines

Balša Šarenac[1][0000-0003-2953-2118] and Stéphane Ducasse[2][0000-0001-6070-6599] and Guillermo Polito[2][0000-0003-0813-8584] and Gordana Rakić[3][0000-0003-1404-4015]

[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
[2] University of Novi Sad, Faculty of Sciences, Trg Dositeja Obradovića 3, 21102 Novi Sad, Serbia
balsasarenac@uns.ac.rs

Abstract. Refactorings are behavior-preserving source code transformations that have become integral to modern Integrated Development Environments (IDEs) and code editors, significantly enhancing software development practices. Our goal is to facilitate the creation of custom refactorings and support users in developing their own. To achieve this, we identify the essential building blocks required for creating composite, modular refactorings.

We propose a set of abstractions that enable the development of a robust refactoring engine. These abstractions include an initial set of transformations, program primitives, and an API for precondition checking and static analysis necessary to create new refactorings. We demonstrate how standard refactorings can be effectively composed using these building blocks, providing concrete examples to illustrate their application.

Keywords: Refactoring, Transformation, Composition, Meta-Model, API.

1 Introduction

Refactorings are behaviour-preserving source code modifications, ensuring that the transformation does not alter the program's external behaviour. In contrast, transformations are non-behavior-preserving source code modifications. Both refactorings and transformations are characterized by preconditions (criteria that must be satisfied for the transformation to execute) and specific source code modifications [1]. Refactorings enhance base transformations with additional behaviour-preserving preconditions and source code modifications, defining them as a subset of transformations. Transformations can also be composed of other transformations, which are referred to as composite transformations.

In our previous work [2], we introduced a new refactoring engine architecture. Our objective is to make this architecture easily extensible and applicable to other dynamically typed object-oriented languages. To achieve this, we define in this paper the core meta-model required to implement transformations and an essential list of transformations necessary for this endeavour.

The meta-model should consist of a minimal set of APIs that must be implemented for the engine to function effectively. There will, of course, be variations and modifications among languages based on differing language semantics. The list proposed here is drawn from our experience developing a new version of Pharo's refactoring engine. Pharo[1] is a pure object-oriented, dynamically typed language and a robust environment. We selected this language because it is relatively simple and features an elegant base meta-model [3]. In addition, the previous version of the refactoring engine was the first real refactoring engine [4, 5].

In the related literature, several papers discuss the meta-models upon which refactorings operate. These papers [6,7] predominantly focus on creating a language-independent meta-model. While this approach has its merits, rendering everything language-independent, followed by the development of language-dependent features, can be considerably challenging and resource-intensive.

Our approach is more streamlined and serves as a foundation for extensibility and adaptability across different languages. A few works [8,9,10,11,21] reference a core set of transformations utilized for the development of other refactorings or transformations, which further validates our focus on foundational building blocks.

The proposed solution comprises an essential set of transformations necessary to (1) establish the foundation of the engine and (2) develop additional transformations based on them. Furthermore, it encompasses the API required for (1) precondition-checking operations and (2) static analysis and includes the essential API for (3) executing the requisite code changes that implement transformations on the source code.

To achieve this goal, we pose several key questions:

- What are the necessary building blocks for creating composite, modular refactorings?
- What is the meta-model required to represent these refactorings?
- What APIs are necessary to express preconditions and generate a change model?

The contribution of this article is a comprehensive framework that includes a set of transformations, a meta-model for representing refactorings, and a programmatic API essential for creating new transformations. We demonstrate how standard refactorings and transformations can be decomposed using these building blocks. This framework provides a reference platform for developing refactoring tools across various programming languages, ensuring ease of use and adaptability.

The remainder of this paper is structured as follows: Section 2 provides a comprehensive overview of the current refactoring engine architecture. Section 3 examines the precondition and static analysis API, while Section 4 details the API for code changes. Section 5 outlines the essential transformations necessary for creating composite refactorings. In Section 6, we present a case study on the extent of reuse of these transformations within the system. Section 7 discusses the significance of using transformations instead of program model primitives, highlighting their advantages and benefits. Related work is discussed in Section 8, while Section 9 concludes the paper.

---

[1] https://pharo.org/

2 Refactoring Engine Architecture

In our previous work [2], we introduced a novel refactoring engine architecture that supports two key scenarios: (1) the interactive application of refactorings/transformations and (2) the scripting of refactorings/transformations. Additionally, our new architecture supports the mixing of transformations and refactorings, with refactorings expressed as a composition of transformations and refactorings. In this article, we focus exclusively on the components related to the building blocks of refactorings/transformations and the underlying meta-models. Figure 1 provides a brief overview of the architecture, concentrating on the engine while excluding user interaction concerns. This low-level architecture inherits from one of the Refactoring Browsers [4, 5].

The architecture consists of:

- Refactoring and Transformation objects that contain the conditions and actions necessary for executing the transformation. The result of their execution generates Change objects within the Namespace entity.
- Conditions represent reified objects that encapsulate precondition-checking logic. They include checks, violators, and error messages, and are executed on Program Model entities.
- Changes represent reified objects that encapsulate the transformation logic within the system. Their existence enables a transactional application of changes. Refactorings are applied to a representation of the program (Program Model). Change entities represent elementary meta-model modifications that can be scoped, cancelled, or filtered by the developer prior to their application.

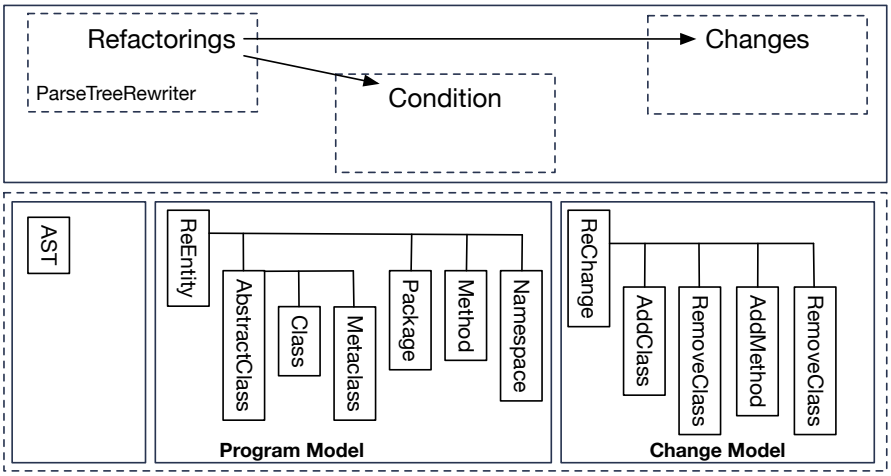

Fig. 1. Overview of the Refactoring Engine Architecture

- Namespace objects allow the encapsulation of a subpart of the system on which specific refactorings are to be executed. By default, refactorings are applied to the entire

system; however, users can narrow the scope to a specific package or a collection of classes.
- The Program Model (Re meta-model) is a collection of classes that partially mimic the system meta-model of the language [3], which is simplified and tailored to meet the requirements of the refactoring engine. They differ in that the Re meta-model is used solely within the Namespace. These classes, along with the namespace, include precondition checking and transformation change-producing APIs.
- The Abstract Syntax Tree (AST) represents the system's meta-model. It is primarily utilised for querying and instantiating the Re meta-model and is mainly focused on representing code manipulation.

In the following sections, we will discuss the key API and core set of transformations.

3 Program Model API for Preconditions and Static Analysis

The Program Model API for preconditions and static analysis primarily includes accessing and querying operations on the meta-model. The complete API is provided in Appendix A. Here, we showcase some examples. The method in Listing 1, is used in the precondition-checking phase to find subclasses of a class that override a method. There are two usages of program model API: allSubclasses and directlyDefinesMethod:.

```
ReHierarchyDefinesMethodCondition >> subclassOf: aClass
redefines: aSelector
   | defs |
   defs := aClass allSubclasses select: [ :each | each
directlyDefinesMethod: aSelector ].
   defs ifNotEmpty: [ violators add: defs. ^ true ].
   ^ false
```

Listing. 1. subclassOf:redefines: method of ReHierarchyDefinesMethodCondition

In Listing 2, there are also two usages of the meta-model: allSubclasses and directlyDefinesInstanceVariable:. This code removes the instance variable from all subclasses that define it before adding that instance variable to the target class.

```
RePullUpInstanceVariableRefactoring >> privateTransform
   class allSubclasses do:
      [:each | (each directlyDefinesInstanceVariable:
variableName)
         ifTrue: [each removeInstanceVariable: variable-
Name]].
   class addInstanceVariable: variableName
```

Listing. 2. privateTransform method of RePullUpInstanceVariableRefactoring

This example serves as a prime illustration of one of the goals of this research. Instead of using the API for code changes directly (i.e., removeInstanceVariable: and addInstanceVariable:), we could utilise existing transformations: Remove Variable and Add Variable. In doing so, we avoid direct interaction with the Program Model API, opting instead for a safer version that includes a check to ensure that removal will not disrupt the existing code. Furthermore, by employing transformations and composition, the resulting code will be more declarative and, consequently, easier to maintain and develop.

4 Program Model API for Code Changes

The Program Model API for code changes includes various operations that generate Change objects. Most of this API is defined within the Namespace. The API includes:

- the creation of new packages and classes,
- compilation of methods,
- creation of instance, class, and pool variables, and
- removal of classes, packages, methods, and variables.

The Re meta-model classes also contain API that is specific to the model class. For instance, the class meta-model includes API for the manipulation of class variables, while the abstract class operates with instance variables and methods. The method meta-model modifies the method itself.

We have already discussed Listing 1, which features two usages of the meta-model: removeInstanceVariable: and addInstanceVariable:. Below, we present additional examples of the usage of this API. In Listing 2, we observe an example of the use of defineClass: from RBNamespace, as well as reparentClasses:to:.

```
ReInsertNewClassRefactoring >> privateTransform
   self model
     defineClass: [ :aBuilder |
      aBuilder
       superclassName: superclass name;
       name: className;
       package: packageName;
       tag: tagName;
       comment: self comment;
       traitComposition: (traitComposition ifNil: #()) ];
      reparentClasses: subclasses to: (self model class-
Named: className)
```

Listing. 3. privateTransform method of ReInsertNewClassRefactoring

In Listing 3, there are usages of AST methods: allChildren, defines:, isSequence, and removeTemporaryNamed:. Additionally, there is a usage of the compileTree: method from RBAbstractClass.

```
ReRemoveTemporaryVariableRefactoring >> privateTransform
   | methodTree modifiedNode |
   methodTree := self definingMethod.
   modifiedNode := (methodTree allChildren select: [:each
| each isSequence])
      detect: [ :sequence | sequence defines: variable-
Name ]
      ifNone: [ nil ].
   modifiedNode ifNotNil: [ modifiedNode removeTempo-
raryNamed: variableName ].
   class compileTree: methodTree
```

Listing. 4. privateTransform method of ReRemoveTemporaryVariableRefactoring

The complete API is displayed in Fig 2.

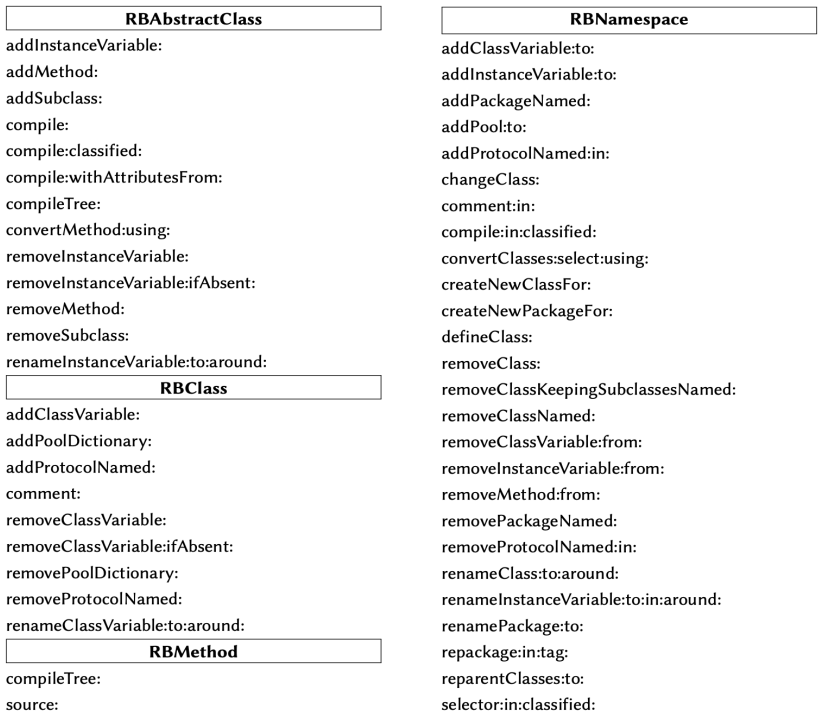

Fig. 2. RBEntity meta-model operations invoked by refactorings and transformations

5 Essential Set of Transformations

In this section, we focus on transformations that are defined using the API from previous sections. Table 1 presents an initial minimal set of transformations. The remaining transformations in the system can be created using these by either specialising (subclassing) or composition.

Table 1. Essential reusable transformations.

Add Method Transformation	Add Variable Transformation
Insert New Class Transformation	Remove Class Transformation
Remove Method Transformation	Remove Variable Transformation
Replace Message Send Transformation	

The core set of transformations consists of those that create and remove program model elements, as well as those that encapsulate the parse tree rewriter.

With this core set of transformations, we can implement most of the refactorings that any refactoring engine supports, such as renaming classes, methods, and variables, adding and removing parameters, and pulling up and pushing down methods and variables. For instance, we can create the Rename Method refactoring by composing the following:

- Add Method transformation that creates a method with the new name.
- Replace Message Send transformation that replaces message sends of the old method with the new method.
- Remove Method transformation that removes the method with the old name.

Similarly, we can create other rename refactorings. However, for more complex transformations, such as those requiring structural modifications, like extracting or inlining methods, we need to delve into AST modifications.

5.1 AST Modification Wrappers

To address the more intricate refactorings that cannot be achieved with the core set of transformations alone, we utilise transformations that wrap AST modifications. These transformations are necessary for operations such as inlining methods or temporaries, extracting blocks into separate temporaries or methods, and refactoring deeply nested code structures.

To facilitate the creation of new refactorings and transformations, we propose the inclusion of the transformations from Table 2. These transformations encapsulate common AST modifications and their accompanying preconditions. Thus, we refer to them as wrappers for AST modifications.

Table 2. Transformations as wrappers around AST modifications.

Add Assignment Transformation	Add Annotation Transformation
Add Message Send Transformation	Add Return Statement Transformation

Add Subtree Transformation	Add Temporary Variable Transformation
Remove Annotation Transformation	Remove Assignment Transformation
Remove Message Send Transformations	Remove Return Statement Transformation
Replace Subtree Transformation	Remove Temporary Variable Transformation

Listing 4 displays an example of the Add Return Statement transformation. This transformation adds a return statement at the end of the method's body. It also includes preconditions that check the validity of this modification, which are omitted for brevity.

```
ReAddReturnStatementTransformation >> privateTransform
   | methodTree messageNode |
   methodTree := self definingMethod.
   messageNode := self parserClass parseExpression: re-
turnValue.
   methodTree body addNode: messageNode.
   self definingClass compileTree: methodTree
```

Listing. 4. privateTransform method from the Add Return Statement transformation.

6 Case Study

In this section, we will evaluate the extent to which these essential transformations are utilised in the system at this point and compare it with the evaluation from the previous paper. The authors in [1] showcased the reuse of the system at that time. Here, we present a similar table that illustrates the current level of reuse in the system, Table 3. Contrary to the previous table, which consisted of commonly used refactorings, we now present the reuse of essential transformations. Comparing this table to the previous one, one can observe an increase in the reuse of Add Method and Remove Method transformations among other refactorings. This increase is attributable to our focus on composition and the reduction of duplication in the code.

Table 3. Essential transformations are core transformations that are used to create other transformations or refactoring. This table shows how much they are used right now.

Transformation	**Used by**
Add Method	Add Variable Accessor, Add Variable Accessor With Lazy Initialization, Children To Siblings, Copy Package, Deprecate Class, Deprecate Method, Duplicate Class, Extract Method, Extract Set Up Method, Generate Equal Hash, Generate Print On, Make Class Abstract, Move Method To Class, Move Method To

	Class Side, Pull Up Method, Push Down Method, Rename Instance Variable
Add Variable	Duplicate Class, Move Instance Variable To Class, Pull Up Variable, Push Down Variable, Split Class
Insert New Class	Children To Siblings, Copy Package, Duplicate Class, Rename And Deprecate Class, Split Class
Remove Class	
Remove Method	Add Parameter, Children To Siblings, Deprecate Class, Inline All Senders, Move Method To Class, Move Method To Class Side, Pull Up Method, Push Down Method, Remove Methods, Remove Parameter, Rename Instance Variable, Rename Method
Remove Variable	Move Instance Variable To Class, Pull Up Variable, Push Down Variable, Split Class
Replace Message Send	Add Parameter, Remove Parameter, Rename Method

It should be noted that AST wrapper transformations do not currently exhibit high usage, as shown in Table 4. This is primarily because these transformations did not exist when the refactorings were initially implemented. During our analysis, we identified several missed reuse opportunities related to these transformations. One of our future goals is to refactor this code to better leverage existing transformations.

Table 4. Reuse of transformations that encapsulate AST modifications.

Transformation	**Used by**
Add Annotation	
Add Assignment	
Add Message Send	
Add Return Statement	
Add Subtree	
Add Temporary Variable	Move Temporary Variable Definition
Remove Annotation	
Remove Assignment	
Remove Return Statement	
Remove Temporary Variable	Move Temporary Variable Definition, Remove Unused Temporary Variables
Replace Subtree	Extract Method

7 Discussion

In this section, we explore the advantages and implications of using transformations over program model primitives in our refactoring framework. By examining specific examples and discussing key design principles, we highlight how transformations can reduce code duplication, promote a more declarative design, and enhance overall safety and reliability.

7.1 Transformation instead of program model primitives

In Listing 2, we observed how addInstanceVariable: and `removeInstanceVariable:` were used directly instead of their corresponding transformations. While this might seem like a more straightforward solution, it leads to code duplication. The same example illustrates the necessity of first checking if a class defines a variable before removing it. This is precisely the precondition of the Remove Variable transformation. Therefore, we could rewrite this to utilise transformations, thereby reducing duplication and promoting a more declarative design. Furthermore, when employing transformations, we do not rely on the Re meta-model and any changes that might affect it, as transformations encapsulate these changes on our behalf.

Another illustrative example of the importance of using transformations instead of program model primitives is removeClassNamed:. This method from the Namespace class removes the class with the specified name. As noted in the previous example, this is one of the more frequently used APIs, resulting in significant duplication surrounding it. More importantly, this method represents one of the more impactful transformations, as improper usage can compromise the system's internals. Thus, it is essential for users to utilise the safer version, the transformation that encapsulates it, providing a secure way to execute it. If our design is implemented correctly, users should not need to access program model APIs directly.

In summary, transformations offer several advantages over using program model primitives:

- Reduced Code Duplication - By encapsulating common operations, we minimise redundant code.
- Promoting Declarative Design - Transformations encourage a more declarative approach to refactoring, making the code easier to understand and maintain.
- Improved Safety and Reliability - By encapsulating complex operations, transformations reduce the risk of errors when using the program model directly.

These benefits make it clear that the adoption of transformations is not merely a stylistic choice but a practical necessity for robust and scalable refactoring systems.

8 Related work

Several authors have worked on developing a language-independent meta-model for refactorings [6], [7]. This approach focuses on creating an independent core, followed by language-specific extensions and user interfaces. In contrast, our approach begins with a language-specific meta-model and progresses towards the language-independent core. We do not intend to create a universal solution; rather, we aim to provide guidelines and foundational building blocks for future researchers and developers who undertake this endeavour.

There is some research on the process of creating composites. Takahashi et al. [12] investigated the decoupling of multiple classes using composite refactorings, and they argue that it is not easy to figure out the right way to do composition by hand. Sousa et al. [13] conducted data mining to analyse composite refactorings. They revealed that many design smells are introduced due to incomplete composite refactorings and demonstrated a correlation between design smells and composite refactorings.

Several authors have researched the composition and implementation of refactorings. Cinnëide [14] focuses on behaviour preservation by creating composite refactorings from primitive refactorings, which have defined pre- and post-conditions. The authors leverage conditions to ensure behaviour preservation. Kniesel-Wünsche et al., [11] introduce a formal model for the automatic, program-independent composition of conditional program transformations, relying on AND and OR operators to compose refactorings. They also supplement the list of base refactorings and preconditions. Compared to our work, their focus is on statically typed languages; thus, their base set of transformations is considerably larger than ours and includes a greater number of program model operation wrappers. Saadeh et al., [15] introduce the concept of fine-grained transformations (FGT), which are identified as steps of other refactorings. The authors describe the approach of creating new refactorings using existing FGTs. They primarily focus on composition and do not disclose the specifics of FGTs. Schäfer et al., [16] and [17] present the implementation of Extract Method refactoring using elementary or atomic refactorings. In [18], the authors adopt a novel approach to refactorings, wherein they do not rely on precondition checking but rather on dependencies. The authors implemented core refactorings based on this notion and compared them with Eclipse's refactorings. We have provided a comparison of our approach with theirs in [19]. Vakilian et al., [9] advocate for the creation of complex refactorings by leveraging the composition of small, predictable changes using a tool, subsequently manually composing them into complex changes. This approach significantly differs from ours, as it does not automate the refactoring process. Ferreira et al., [10] present a list of refactorings specific to React, JavaScript, CSS, and traditional ones. This approach highlights domain-specific refactorings and places greater emphasis on that aspect, whereas this paper focuses on composition and base transformations. Li et al., [20] provide users with a template- and rule-based program transformation and analysis API to enable them to define custom refactorings in Erlang. This approach is similar to ours but is tailored to functional languages; thus, many of the concepts do not apply to our use case.

Several authors have leveraged composition to facilitate the introduction of design patterns. Ajouli et al. [21] and [22] present two papers on the introduction of the visitor design pattern, focusing on the composition of smaller refactorings. In [21], there is a comprehensive list of refactorings and their preconditions that are utilised to create a visitor and other refactorings. Kim et al. [23] introduce Reflected Refactoring, a Java package that automates the creation of classical design patterns within the codebase. This serves as a classic example of how the composition of elementary refactorings empowers the creation of new custom and powerful refactorings.

9 Conclusion

In this paper, we aim to provide a detailed framework that can serve as a reference for anyone developing their own refactoring engine. We presented a comprehensive framework for developing an extensible refactoring engine applicable to dynamically typed object-oriented languages such as Pharo. The key contributions include:

- A Set of Transformations: We identified a core set of transformations necessary for building modular and composite refactorings.
- A Meta-Model: We outlined a formal representation that captures the structure and behaviour of these refactorings.
- A Programmatic API: We provided an interface to facilitate the validation and creation of changes.

Our framework is demonstrated through examples in Pharo. We present API usage examples and tables with transformations and APIs, ensuring that the approach remains language-independent where possible.

In conclusion, this work lays a solid foundation for building powerful and flexible refactoring engines. The detailed specifications in our paper provide a comprehensive guide for future development, ensuring both flexibility and robustness. Future research should aim to further enhance the framework's capabilities by migrating refactorings to a fully composite architecture. Additionally, we intend to develop a user-friendly interface that allows non-experts to create composite refactorings. Future work could explore additional use cases and refine the transformation framework for improved usability and effectiveness.

10 Acknowledgment

This research has been supported by the Ministry of Science, Technological Development and Innovation (Contract No. 451-03-137/2025-03/200156) and the Faculty of Technical Sciences, University of Novi Sad through project"Scientific and Artistic Research Work of Researchers in Teaching and Associate Positions at the Faculty of Technical Sciences, University of Novi Sad2025" (No. 01-50/295).

References

1. Anquetil, N., Campero, M., Ducasse, S., Sandoval, J.-P., Tesone, P.: Transformation-based refactorings: a first analysis. In: International Workshop of Smalltalk Technologies, IWST'22 (2022).
2. Šarenac, B., Anquetil, N., Ducasse, S., Tesone, P.: A new architecture reconciling refactorings and transformations. Journal of Computer Languages (2024) 101273. https://doi.org/10.1016/j.cola.2024.101273.
3. Thomas, I., Ducasse, S., Tesone, P., Polito, G.: Pharo: a reflective language - analyzing the reflective API and its internal dependencies. Journal of Computer Languages (2024). https://doi.org/10.1016/j.scico.2014.02.016.
4. Roberts, D., Brant, J., Johnson, R.E.: A refactoring tool for Smalltalk. Theory and Practice of Object Systems 3, 253–263 (1997).
5. Roberts, D., Brant, J., Johnson, R.E., Opdyke, B.: An automated refactoring tool. In: Proceedings of ICAST '96, pp. 1–5 (1996).
6. Strein, D., Kratz, H., Lowe, W.: Cross-language program analysis and refactoring. In: 2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation, pp. 207–216. IEEE, (2006). https://doi.org/10.1109/SCAM.2006.10.
7. Tichelaar, S., Ducasse, S., Demeyer, S., Nierstrasz, O.: A meta-model for language-independent refactoring. In: Proceedings of International Symposium on Principles of Software Evolution, ISPSE'00, pp. 157–167. IEEE Computer Society Press, (2000). https://doi.org/10.1109/ISPSE.2000.913233.
8. Werner, M.M.: Facilitating Schema Evolution With Automatic Program Transformation. Ph.D. thesis, Northeastern University (1999).
9. Vakilian, M., Chen, N., Moghaddam, R.Z., Negara, S., Johnson, R.E.: A compositional paradigm of automating refactorings. In: European Conference on Object-Oriented Programming, pp. 527–551 (2013).
10. Ferreira, F., Borges, H.S., Valente, M.T.: Refactoring React-based web apps. Journal of Systems and Software 215, 112105 (2024). https://doi.org/10.1016/j.jss.2024.112105.
11. Kniesel-Wünsche, G., Koch, H.: Static composition of refactorings. Science of Computer Programming 52, 9–51 (2004). https://doi.org/10.1016/j.scico.2004.03.002.
12. Takahashi, Y., Nitta, N.: Composite refactoring for decoupling multiple classes. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 594–598. IEEE, (2016). https://doi.org/10.1109/SANER.2016.54.
13. Sousa, L., Cedrim, D., Garcia, A., Oizumi, W., Bibiano, A.C., Oliveira, D., Kim, M., Oliveira, A.: Characterizing and identifying composite refactorings: Concepts, heuristics and patterns. In: 2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR), pp. 186–197 (2020). https://doi.org/10.1145/3379597.3387477.
14. Ò Cinnéide, M.: Composite refactorings for Java programs. In: Proceedings of the Workshop on Formal Techniques for Java Programs, European Conference on Object-Oriented Programming (2000).
15. Saadeh, E., Kourie, D.G.: Composite refactoring using fine-grained transformations. In: Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists, SAICSIT '09, pp. 22–29. ACM, New York, NY, USA (2009). https://doi.org/10.1145/1632149.1632154.
16. Schäfer, M.: Specification, Implementation and Verification of Refactorings. Ph.D. thesis, Oxford University Computing Laboratory (2010).
17. Schäfer, M., Verbaere, M., Ekman, T., de Moor, O.: Stepping stones over the refactoring rubicon – lightweight language extensions to easily realise refactorings. In: Drossopoulou,

S. (ed.) European Conference on Object-Oriented Programming (ECOOP), pp. 369–393. Springer-Verlag, (2009).
18. Schäfer, M., de Moor, O.: Specifying and implementing refactorings. In: Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'10), pp. 1–15 (2010).
19. Šarenac, B., Ducasse, S., Polito, G., Rakić, G.: Modular and extensible extract method. In: International Workshop on Smalltalk Technologies (2024).
20. Li, H., Thompson, S.: A User-extensible Refactoring Tool for Erlang Programs. Technical report, University of Kent (2011). https://kar.kent.ac.uk/30720/.
21. Ajouli, A., Cohen, J.: Refactoring composite to visitor and inverse transformation in Java. ArXiv abs/1112.4271 (2011). https://api.semanticscholar.org/CorpusID:14666606.
22. Cohen, J., Ajouli, A.: Practical use of static composition of refactoring operations. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, pp. 1700–1705. ACM, New York, NY, USA (2013). https://doi.org/10.1145/2480362.2480684.
23. Kim, J., Batory, D., Dig, D.: Scripting parametric refactorings in Java to retrofit design patterns. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 1–5 (2015).

Appendix A Program Model API for Precondition Checking and Static Analysis

RBAbstractClass
=
allClassVariableNames
allInstanceVariableNames
allMethods
allPoolDictionaryNames
allSelectors
allSubclasses
allSubclassesDo:
allSuperclasses
allSuperclassesUntil:
binding
bindingOf:
canonicalArgumentName
canUnderstand:
classBinding
classSide
classVariableNames
checkSelector:using:
definesClassVariable:
definesInstanceVariable:
definesMethod:
definesPoolDictionary:
definesVariable:
directlyDefinesClassVariable:
directlyDefinesLocalMethod:
directlyDefinesInstanceVariable:
directlyDefinesMethod:
directlyDefinesPoolDictionary:
directlyDefinesVariable:
existingMethodsThatReferTo:
existingMethodsThatReferToClassVariable:
existingMethodsThatReferToInstanceVariable:
existingMethodsThatReferToSharedVariable:
existingSelectorsThatReferToClassVariable:
existingSelectorsThatReferToInstanceVariable:
existingSelectorsThatReferToSharedVariable:
firstSuperclassRedefines:
getterMethodFor:
hash
hasRemoved:
hierarchyDefinesClassVariable:
hierarchyDefinesInstanceVariable:
hierarchyDefinesMethod:
hierarchyDefinesPoolDictionary:
hierarchyDefinesVariable:
includesClass:
initialize
instanceVariableNames
instanceVariableNames:
instanceSide
isAbstract
isDefined
isMeta
localMethods
localSelectors
methodFor:
model
model:
name
newMethods
parseTreeFor:
parseTreeForSelector:
parseTreeSearcherClass
privateInstanceVariableNames
protocols
protocolsFor:
realClass
realName:
removedMethods
selectors
setterMethodFor:
soleInstance
sourceCodeFor:
subclasses
subclassesDo:
subclassRedefines:
superclass
superclassRedefines:
theMetaClass
theNonMetaClass
typeOfClassVariable:
whoDefinesClassVariable:
whoDefinesInstanceVariable:
withAllSubclasses
withAllSuperclasses
withAllSuperclassesUntil:
whichClassIncludesSelector:
whichMethodsReferToInstanceVariable:
whichMethodsReferToSharedVariable:
whichSelectorsReferToClass:
whichSelectorsReferToClassVariable:
whichSelectorsReferToInstanceVariable:
whichSelectorsReferToSharedVariable:
whichSelectorsReferToSymbol:
whoDefinesMethod:

RBClass

allClassVariableNames
allPoolDictionaryNames
classPool
classVariableNames
classVariableNames:
comment
directlyDefinesClassVariable:
directlyDefinesPoolDictionary:
hasSubclasses
initialize
instanceSide
isEmptyClass
isManifest
isMeta
isSelfEvaluating
isSharedPool
methods
methodsUsingClassVariableNamed:
packageName
poolDictionaryNames
privateClassVariableNames
privatePoolDictionaryNames
sharedPools
sharedPoolNames
tagName
traitComposition
traitComposition:

RBMetaclass

allClassVariableNames
allPoolDictionaryNames
classSide
classVariableNames
directlyDefinesClassVariable:
directlyDefinesPoolDictionary:
isManifest
isMeta
realName:

RBPackage

model
model:
name
name:
realPackage
realPackage:
realPackageFor:

RBTrait

definesInstanceVariable:
isTrait
superclass
users

RBNamespace

allClassesDo:
allClassesInPackages:do:
allImplementorsOf:
allImplementorsOf:do:
allImplementorsOf:inPackages:
allReferencesTo:
allReferencesTo:do:
allReferencesTo:in:
allReferencesTo:inPackages:
allReferencesTo:inPackages:do:
allReferencesToClass:do:
allReferencesToClass:inPackages:do:
changes
changeClass:
classesReferencingClass:
classFor:
classNamed:
classNameFor:
classObjectFor:
description
description:
environment
environment:
hasCreatedClassFor:
hasPackageRemoved:
hasRemoved:
includesClassNamed:
includesGlobal:
includesPackageNamed:
initialize
metaclassNamed:
methodsReferencingClass:
name
name:
packageNamed:
privateImplementorsOf:
privateImplementorsOf:in:
privateReferencesTo:
privateReferencesTo:in:
privateReferencesTo:inPackages:
privateRootClasses
rootClasses

RBMethod

ast
argumentNames
compiledMethod
isFromTrait
isFromTrait:
literal:containsReferenceTo:
method
methodClass
modelClass
modelClass:
origin
package
parseTree
parserTreeSearcher
parseTreeSearcherClass
protocols
refersToClassNamed:
refersToSymbol:
refersToVariable:
selector
source