The Three-Step Refactoring Detector Pattern

Anastasios Tsimakis
Department of Computer Science and Engineering, University of Ioannina, Greece
atsimakis@gmail.com

Apostolos V. Zarras
Department of Computer Science and Engineering, University of Ioannina, Greece
zarras@cs.uoi.gr

Panos Vassiliadis
Department of Computer Science and Engineering, University of Ioannina, Greece
pvassil@cs.uoi.gr

ABSTRACT

The development of a tool that detects opportunities for refactoring in source code is not an easy task. Our experience in developing such a tool revealed the Three-Step Refactoring Detector pattern. The main idea behind the pattern is to develop an extensible hierarchy of refactoring detectors, with respect to a general three-step refactoring detection process. The proposed pattern facilitates the expansion of the hierarchy with new refactoring detectors and enables the reuse of existing refactoring detectors, provided by third party developers. Concerning maintainability, the pattern promotes the development of simple, clean and technology independent refactoring detectors. We have used the pattern for the development of 11 different refactoring detectors in the context of our tool. The pattern has not been observed in other contexts. However, the usage of the pattern in our tool brought out specific empirical evidence of its benefits, which we discuss in this paper.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**;

KEYWORDS

Refactoring, Refactoring Detection, Patterns

ACM Reference Format:
Anastasios Tsimakis, Apostolos V. Zarras, and Panos Vassiliadis. 2019. The Three-Step Refactoring Detector Pattern. In *24th European Conference on Pattern Languages of Programs (EuroPLoP '19), July 3–7, 2019, Irsee, Germany.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/XXXXXXXX.XXXXXXXX

1 INTRODUCTION

Back in the 90's, Opdyke introduced refactoring as a behavior preserving process that changes a software, so as to enable other changes to be made more easily [13]. Since then, several refactoring approaches have been proposed in many different domains [3, 11].

Concerning OO software, Martin Fowler introduced a well-known catalog of refactorings [6]. These are also known as floss refactoring or micro-refactoring, referring to small, individual changes in the source code rather than a large-scale, organised effort to restructure the entirety - or a large part - of it [10, 12], and will be what we will henceforth refer to as "refactoring".

In the same context, several techniques that detect refactoring opportunities have been proposed. Typically, these techniques detect opportunities for refactoring, based on quality metrics or measurable properties that reveal certain code smells and/or bad practices. This allows the creation of tools for automated and reliable recognition of refactoring opportunities, covering a wide range of refactorings (e.g., moving methods [2, 4, 15, 18, 19], extracting methods [16, 20], extracting classes [1, 5], extracting interfaces [9, 14]). Such tools may not only notify the programmer of an opportunity to refactor, but also automatically make the necessary alterations to the code to facilitate the refactoring's completion/realization. Recently, Al Dallal [3] performed an extensive survey that focuses on techniques and tools for the detection of refactoring opportunities. This effort indicates that the interest in refactoring is constantly growing. At the same time, the state of the art on refactoring tools and techniques is vast, including a large variety of approaches that study and facilitate the overall refactoring process.

However, as the refactoring field grows, the development of new refactoring tools becomes more complex, involving more advanced design concerns that should be addressed. In particular, an important issue that is brought out by Kim et al. [8] is the need to combine refactorings in more complex evolution tasks. To deal with this issue, the next generation of refactoring tools should be *extensible* and *maintainable*. So far, these concerns are not really addressed. In particular, many tools focus on a single refactoring. In Al Dallal's survey, for instance, only 25% of the refactoring detection tools concern more than two refactorings. Even in these tools, extensibility, and maintainability are not primary concerns, because typically the tools focus on a fixed set of refactorings.

On our side, we have faced the aforementioned concerns in the development of the Refactoring Trip Advisor, a tool that provides actionable recommendations regarding which refactoring(s) to use before, after, or instead of a particular refactoring. The tool further provides guidelines on how to apply refactorings, and enables the detection of refactoring opportunities. The issues that we encountered in the design and implementation of the Refactoring Trip Advisor brought out an interesting pattern that can help other toolmakers to develop their own refactoring detection tools, regardless of whether these will be incorporated in a specific IDE or editor, or will be completely independent/standalone.

The Three-Step Refactoring Detector pattern *facilitates the development of refactoring detectors, via a polymorphic hierarchy of template classes that realize a general three-step refactoring detection process.* This structure allows a tool-maker to easily *add new*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroPLoP '19, July 3–7, 2019, Irsee, Germany
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN XXXXXXXX…$15.00
https://doi.org/10.1145/XXXXXXXX.XXXXXXXX

refactoring detectors to his tool, and *reuse existing refactoring detectors* that are provided by third-party developers. From a broader perspective, the pattern promotes the development of simple, clean and technology independent refactoring detectors that can be *easily maintained*. We consider this solution as a pattern because we have used it for the development of 11 different refactoring detectors in the context of Refactoring Trip Advisor. The pattern has not (yet) been observed elsewhere. However, the usage of the pattern in our project revealed specific empirical evidence of its advantages, which we report in this paper.

The rest of this paper is structured as follows. In Section 2, we introduce the main concepts of the Three-Step Refactoring Detector pattern and we discuss the use of the pattern in the Refactoring Trip Advisor. In Section 3, we summarize our contribution.

2 THREE-STEP REFACTORING DETECTOR

Context

A tool-maker is building a stand-alone refactoring tool, or a refactoring plugin for an existing IDE or editor. As part of the tool, the tool-maker wants to develop a set of refactoring detectors.a

Problem

IDE users' needs change and there are advances in the field of refactoring. Thus, the design of the refactoring tool should make future changes easier.

Forces

- Developing in one shot a refactoring tool that supports all the refactorings that the users may possibly need is not pragmatic.
- To deal with the users' evolving requirements, it should be possible to extend the tool with new refactoring detectors.
- It is not practical to implement from scratch all the refactoring detectors for every refactoring that the users may possibly need.
- To exploit the state of the art on refactoring, the tool should not only include refactoring detectors developed by the tool-maker (in-house detectors), but also reuse third-party refactoring detectors, provided by other developers.
- The addition of new refactoring detectors and/or the reuse of pre-existing refactoring detectors shall be easier if the overall implementation of the tool is clean, simple, and free from code smells.
- The overall detection process of refactoring opportunities is similar for different refactorings, as it involves tasks that (1) limit the search space to the parts of the project the user is interested in, (2) select the particular project elements that may contain refactoring opportunities and (3) detect the actual refactoring opportunities, respectively.
- Developing refactoring detectors independently from each other may introduce unnecessary complexity and code duplication in the implementation of the tool, due to the re-implementation of common refactoring detection tasks over and over again.

Solution

The key idea of the solution is that all the refactoring detectors implement the same general refactoring detection process that consists of the following 3-steps:

(1) Identify the refactoring scope: When a developer is looking for refactoring opportunities he may be interesting in searching in specific parts of his/her project. Hence, in the first step of the process the developer instructs the refactoring detector on which project elements to focus on - these elements constitute the ***refactoring scope***.
(2) Identify the refactoring subjects: in second step, the refactoring detector identifies within the elements of the refactoring scope the ***refactoring subjects***, i.e., the smallest self-contained elements that can contain refactoring opportunities.
(3) Identify the refactoring opportunities: in the third step, the refactoring detector identifies within the refactoring subjects the ***refactoring opportunities***, i.e., the specific elements that can be refactored.

Then, each individual refactoring detector customizes the steps of the refactoring detection process appropriately, according to the specificities of the specific refactoring that is tackles. The different variants of the general refactoring detection process result in *a polymorphic hierarchy of template classes* that is discussed in detail in the pattern structure section that follows.

Structure

Figure 1, depicts the structure of the pattern. In particular, the key classes of the polymorphic hierarchy are `RefactoringDetector`, `RefactoringDetectorForClasses`, and `RefactoringDetectorForMethods`. `ProjectExplorer` provides information concerning the user interaction with the refactoring tool. `Project`, provides information about the structure of the project that the user is working on. We assume a typical project structure, consisting of packages that comprise classes, and so on. Note that typically the implementation of classes like `ProjectExplorer` and `Project` is technology dependent, varying with respect to specific IDEs, APIs, etc. Consequently, in the pattern structure these classes can be thought of as wrappers or facades that isolate the refactoring tool from the underlying technologies and hide the complexity that emerges from the usage of these technologies.

In more detail, `RefactoringDetector` is the abstract class that realizes the general three-step refactoring detection process. The starting point is the `execute()` method (Figure 2(top-left)). The method executes the three steps of the refactoring detection process, by calling respective methods.

- `identifyScope()` implements the first step of the process (Figure 2 (top-right)). This step is common for all the refactoring detectors. Specifically, every refactoring detector identifies the ***refactoring scope***.
  - The refactoring scope may be the whole project, a specific package, class, method, etc. Typically, the user of the refactoring tool selects the refactoring scope using the project explorer.

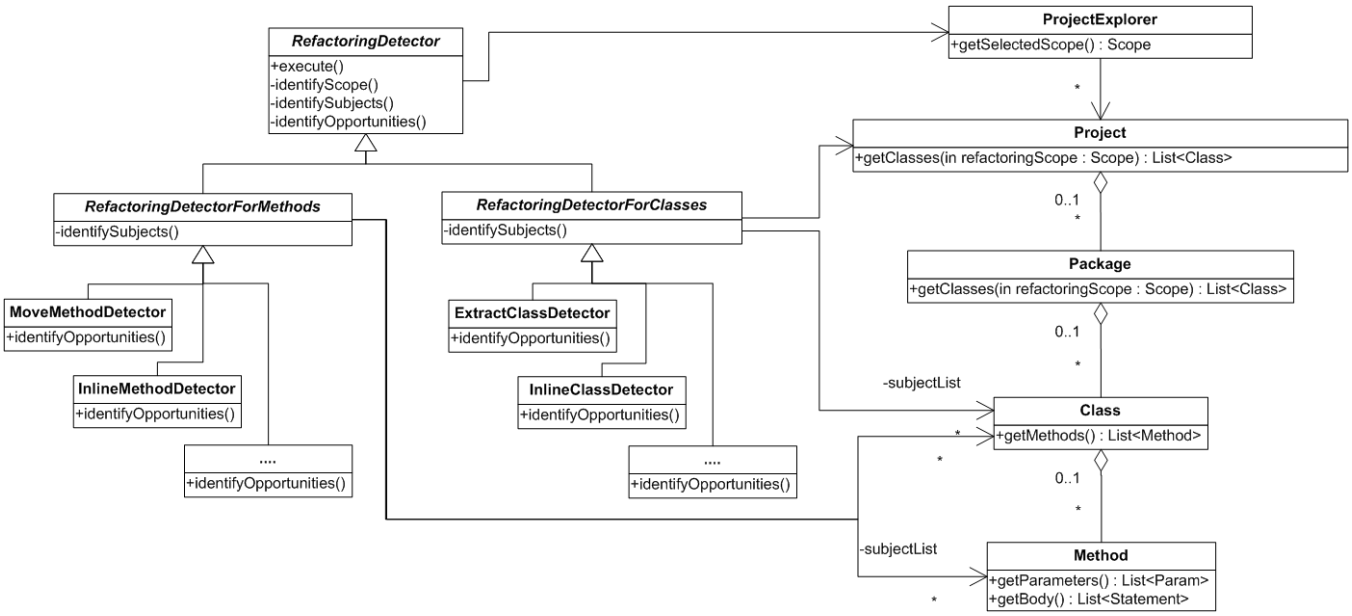


Figure 1: Pattern structure.

  - In this step, the refactoring detector further checks if the selected scope is valid, based on a list of valid scopes. Although the first step of the refactoring process is common for all the refactoring detectors, the list of valid scopes varies, depending on the refactorings. For instance, several refactorings concern classes. So, for these refactorings it does not make sense to select a method as a refactoring scope. In general, to be valid the refactoring scope can not be more narrow than the refactoring subjects (more details in the following).
- `identifySubjects()` realizes the second step of the process. In this step, each refactoring detector looks for a list of ***refactoring subjects*** that belong to the selected refactoring scope.
  - Tables 1 to 6, concern Fowler's catalog of refactorings [6]. Each table corresponds to a different category of refactorings, as specified in Fowler's catalog. For each refactoring, we specify the particular type of subjects where the refactoring can be applied to. In general, we can distinguish two different types of refactorings, those that can be applied to *methods*, and those that can be applied to *classes*.
  - Consequently, in the second step of the process, the refactoring detectors of the first type identify as refactoring subjects a *list of methods*, while the refactoring detectors of the second type identify as refactoring subjects a *list of classes*.
  - Thus, in the pattern structure there are two different subclasses of `RefactoringDetector`, namely `RefactoringDetectorForClasses` and `RefactoringDetectorForMethods`. Moreover, `identifySubjects()` is an abstract method of the `RefactoringDetector` class. The subclasses of `RefactoringDetector` provide the respective concrete implementations of `identifySubjects()` (Figure 2(bottom)).
- `identifyOpportunities()` implements the third step of the process. This step is different for every refactoring detector. In particular, every refactoring detector identifies the actual ***refactoring opportunities*** *within the refactoring subjects that fulfil all the criteria for the refactoring in question to be applied.* The type of the identified refactoring opportunities depends on the particular refactoring (Tables 1 to 6). The criteria for detecting refactoring opportunities depend on the particular technique that is employed for the detection (e.g., a threshold for a particular complexity metric, the parameters of a clustering algorithm, etc.). Therefore, `identifyOpportunities()` is an abstract method of the `RefactoringDetector` class. The concrete implementations of the abstract method are provided by different refactoring specific subclasses of `RefactoringDetectorForClass` or `RefactoringDetectorForMethods`.

Examples

Refactoring Detectors for Methods. Inline Method and Replace Temp with Query are two entirely different refactorings. However, their common characteristic is that they are both applied to methods (Table 1).

More specifically, Inline Method concerns small-sized methods, whose bodies are as clear as the methods themselves [6]. The bodies of such methods can be put in the bodies of the methods' callers, without making them much more complex. Then, the short methods can be removed.

On the other hand, Replace Temp with Query concerns methods with local variables that are assigned once, with the result of a

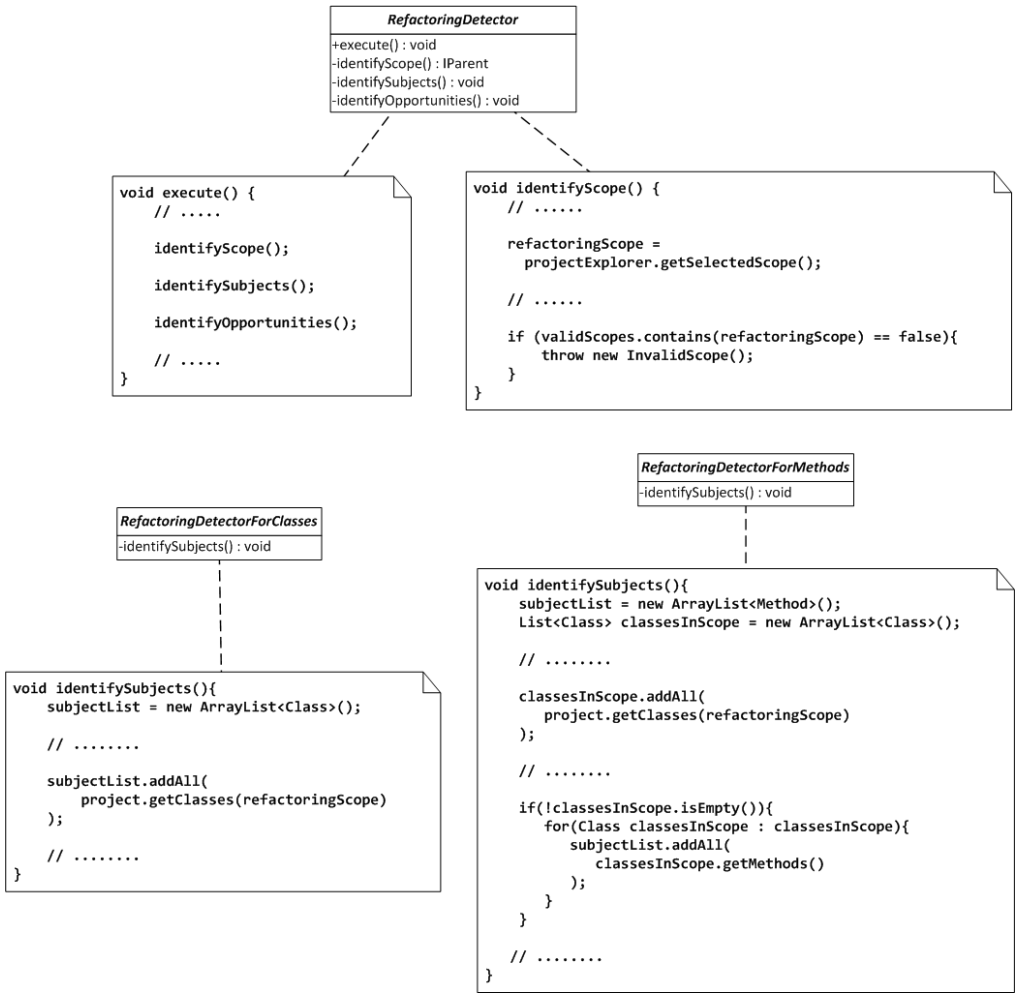


Figure 2: Pattern participants.

Table 1: Composing Methods.

Refactoring	Subject Type	Opportunity Type
Extract Method	Method	Set of Statements
Inline Method	Method	Method
Inline Temp	Method	Variable
Replace Temp with Query	Method	Variable
Introduce Explaining Variable	Method	Expression
Split Temporary Variable	Method	Variable
Remove Assignments to Parameters	Method	Parameter
Replace Method with Method Object	Method	Method
Substitute Algorithm	Class	Set of Statements

Table 2: Moving Features Between Objects.

Refactoring	Subject Type	Opportunity Type
Move Method	Method	Method
Move Field	Class	Field
Extract Class	Class	Set of Statements
Inline Class	Class	Class
Hide Delegate	Class	Class
Remove Middle Man	Class	Class
Introduce Foreign Method	Class	Method
Introduce Local Extension	Class	Class

particular expression [6]. The idea is to extract the expression into a method and replace all references to the variable with calls to the extracted method.

To implement refactoring detectors for the aforementioned refactorings we create respective classes that extend the `RefactoringDetectorForMethods` class (Figure 3).

`InlineMethodDetector` detects Inline Method opportunities (Figure 3). To this end, the detector iterates through the list of refactoring subjects. It checks the size of each subject method. If

Table 3: Simplifying Conditional Expressions.

Refactoring	Subject Type	Opportunity Type
Decompose Conditional	Method	Conditional Statement
Consolidate Conditional Expression	Method	Set of Statements
Consolidate Duplicate Conditional Fragments	Method	Set of Statements
Remove Control Flag	Method	Variable
Replace Conditional with Guard Clauses	Method	Set of Statements
Replace Conditional with Polymorphism	Method	Set of Statements
Introduce Null Object	Method	Conditional Statement
Introduce Assertion	Method	Conditional Statement

Table 4: Organizing Data.

Refactoring	Subject Type	Opportunity Type
Self Encapsulate Field	Class	Field
Replace Data Value with Object	Class	Set of Statements
Change Value to Reference	Class	Field
Change Reference to Value	Class	Class
Replace Array with Object	Class	Set of Statements
Duplicate Observed Data	Class	Set of Statements
Change Unidirectional Association to Bidirectional	Class	Class
Change Bidirectional Association to Unidirectional	Class	Class
Replace Magic Number with Symbolic Constant	Method	Set of Statements
Encapsulate Field	Class	Field
Encapsulate Collection	Class	Field
Replace Record with Data Class	Class	Set of Statements
Replace Type Code with Class	Class	Field
Replace Type Code with Subclasses	Class	Field
Replace Tyle Code with State/Strategy	Class	Field
Replace Subclass with Fields	Class	Class

Table 5: Making Method Calls Simpler.

Refactoring	Subject Type	Opportunity Type
Rename Method	Method	Method
Add Parameter	Method	Method
Remove Parameter	Method	Parameter
Seperate Query from Modifier	Method	Method
Parameterize Method	Class	Method
Replace Parameter with Explicit Methods	Method	Parameter
Preserve Whole Object	Method	Set of Parameters
Replace Parameter with Method	Method	Set of Parameters
Introduce Parameter Object	Class	Set of Parameters
Introduce Parameter Object	Method	Method
Remove Setting Method	Class	Method
Hide Method	Method	Method
Replace Constructor with Factory Method	Class	Method (Constructor)
Encapsulate Downcast	Method	Method
Replace Error Code with Exception	Method	Method
Replace Exception with Test	Method	Set of Statements

Table 6: Dealing with Generalization.

Refactoring	Subject Type	Opportunity Type
Pull Up Field	Class	Field
Pull Up Method	Class	Method
Pull Up Constructor Body	Class	Method
Push Down Method	Method	Method
Push Down Field	Class	Field
Extract Subclass	Class	Set of Statements
Extract Superclass	Class	Set of Statements
Extract Interface	Class	Set of Statements
Collapse Hierarchy	Class	Class
Form Template Method	Class	Set of Statements
Replace Inheritance with Delegation	Class	Class
Replace Delegation with Inheritance	Class	Class

the size of the subject method is less than `theta` statements (`theta` is a given threshold) and the method is not an accessor, a mutator, or a predicate, the detector adds the method to the list of refactoring opportunities.

`ReplaceTempWithQueryDetector` identifies Replace Temp with Query opportunities (Figure 3). In particular, the detector iterates through the list of refactoring subjects. If a subject method contains local variables that are assigned once, the detector includes those variables in the list of refactoring opportunities.

Refactoring Detectors for Classes. Move Method and Extract Class two different refactorings that concern classes (Table 2).

Move Method is applied to classes that have misplaced methods. Specifically, if a method is using (respectively used by) more features of another class, then the method can be moved to that other class.

Extract Class is applied to classes that have many responsibilities. To refactor a class that is doing too much work we have to create new classes for the different responsibilities of the class and move the relevant fields and methods from the class into the new classes.

To detect opportunities for these refactorings we can reuse third-party refactoring detectors provided by the JDeodorant refactoring framework. The JDeodorant Move Method detector relies on a Jaccard-like metric that measures the distance between methods and classes [19]. The distance between a method and a class is small if the method is using many of the class entities (methods and attributes). Based on the values of the distance metric, the detector identifies candidate methods to be moved to other classes. The JDeodorant Extract Class detector relies on agglomerative clustering [5] to regroup the methods and fields of a particular class. The algorithm decides the placement of the class entities based on a Jaccard-like distance metric.

To reuse the aforementioned third-party detectors we extend the `RefactoringDetectorForClasses` class (Figure 4). In particular, the `MoveMethodDetector` is a facade for the JDeodorant Move Method detector, while the `ExtractClassDetector` is a facade for the JDeodorant Extract Class detector.

Consequences

Benefits:

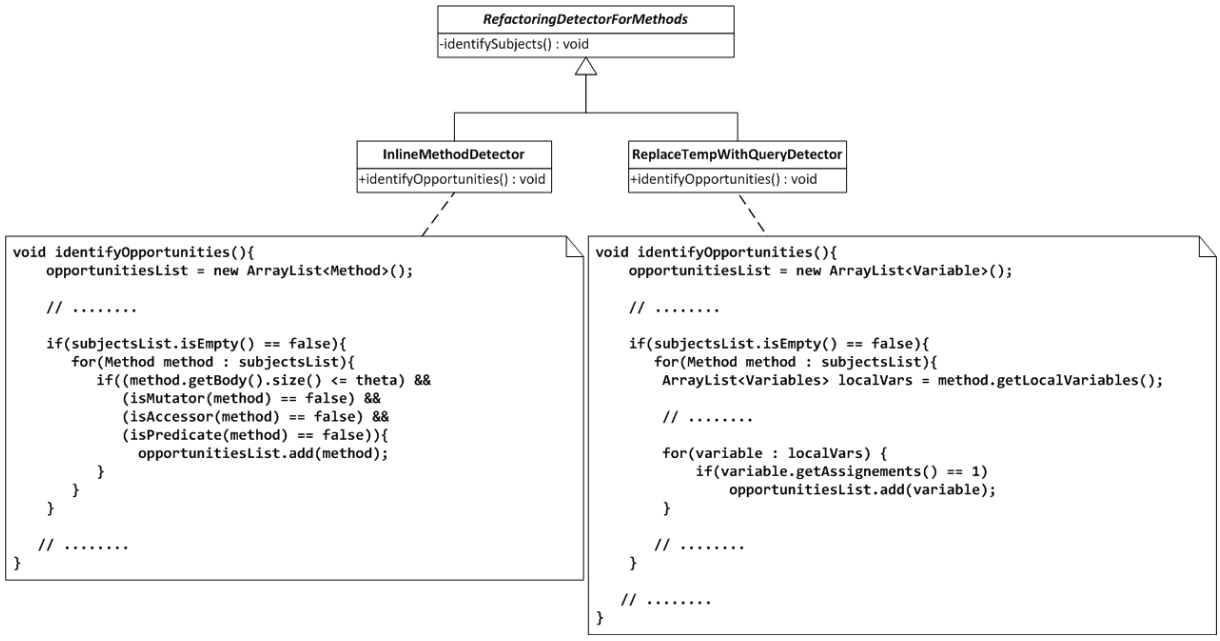


Figure 3: Refactoring detectors for methods.

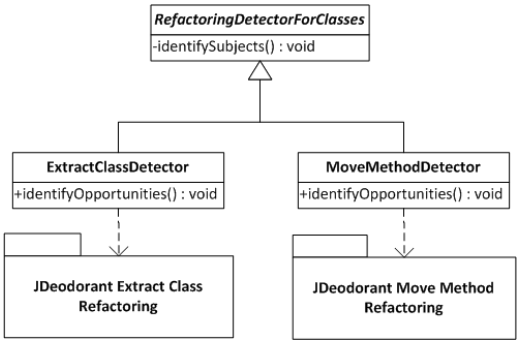


Figure 4: Refactoring detectors for classes.

- *Extensibility*: The pattern makes the addition of new refactoring detectors easy. Specifically, to add an new refactoring detector the tool-maker has to add a new subclass in the polymorphic hierarchy that provides a concrete implementation of the `identifyOpportunities()` method.
- *Reuse*: Similarly, the pattern facilitates the reuse of existing refactoring detectors. To reuse an third-party refactoring detector, the tool-maker has to add a new subclass of `RefactoringDetectorForClasses` or `RefactoringDetectorForMethods` that serves as a facade or an adaptor of the functionality that is provided by the third-party refactoring detector.
- *Maintainability*: The polymorphic hierarchy of template classes promotes the clear separation of responsibilities and facilitates the avoidance of code duplication. The pattern structure comprises specific elements that isolate the hierarchy from underlying technologies used for the interaction between the user and the refactoring tool, the parsing of a software project, the navigation through the structure of the software project, and so on.

Liabilities:

- To add new refactoring detectors, or reuse detectors developed by third-party developers the tool-maker has to understand the basic structure of the pattern. This could be an issue if they are not familiar with the notions of polymorphism and design patterns.
- It is not possible to use the pattern to develop a refactoring tool, with a programming language that does not support polymorphism.
- The functionality of refactoring detectors developed by third-party developers is not directly used. On the contrary, the detectors are integrated with the refactoring tool via facades or adaptors, which may introduce an additional performance or memory overhead.

Related Patterns

Three-Step Refactoring Detector is related to Template Method [7]. The idea behind Template Method is to have a method in a base class that defines the skeleton of an algorithm, as a sequence of steps. Some of the steps correspond to abstract methods. Therefore, their definition is deferred to concrete subclasses of the base class. Three-Step Refactoring Detector adapts Template Method to the specificities of the refactoring detection problem, as its core concept is the polymorphic hierarchy of template classes that define the general three-step refactoring process.

Table 7: Refactoring detectors in the Refactoring Trip Advisor.

Refactoring Detector	Origin
InlineMethodIdentification	In House
InlineTempIdentification	
IntroduceExplainingVariableIdentification	
IntroduceParameterObjectIdentification	
RemoveAssignmentsToParametersIdentification	
ReplaceMethodWithMethodObjectIdentification	
ReplaceTempWithQueryIdentification	
SplitTemporaryVariableIdentification	
ExtractClassIdentification	Third-party (JDeodorant)
ExtractMethodIdentification	
MoveMethodIdentification	

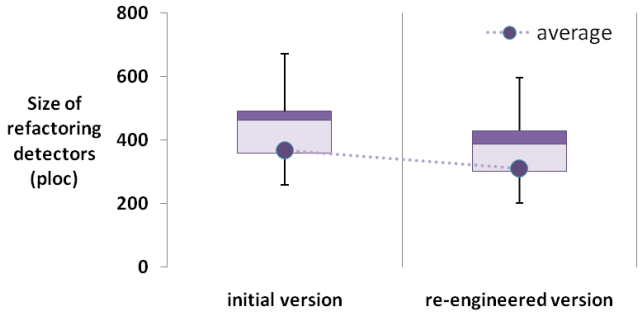


Figure 6: Comparing the statistical distributions of sizes for the refactoring detectors in the initial and the re-engineered versions of the Refactoring Trip Advisor.

Three-Step Refactoring Detector is further related with Adapter and Facade [7]. On the one hand, Adapter facilitates the integration of incompatible classes, by converting the interface of one class into an interface expected by the classes that use it. On the other hand, the idea behind Facade is to provide an convenient interface for a subsystem, to make it more easy to use. In the case of Three-Step Refactoring Detector, both patterns can be used to extend the polymorphic hierarchy of template classes with third-party refactoring detectors.

Empirical Evidence

We employed the Three-Step Refactoring Detector pattern to improve the design and implementation of the Refactoring Trip Advisor [21].

The Refactoring Trip Advisor has two main goals:

- To provide the theoretical background of each refactoring in order to assist the user to understand its importance and use.
- To automatically detect code blocks that are applicable for refactoring.

To go into more detail, the Refactoring Trip Advisor contains a visual map of 68 refactorings, introduced in Fowler's catalog. The map is represented as a graph, with nodes corresponding to refactorings and edges, depicting relations between refactorings. There are three types of relations between refactorings: *succession*, *part-of*, and *instead-of*, denoting respectively that a refactoring could be applied after another, a refactoring is combined with others to form a more complex, composite refactoring, and that a refactoring is an alternative option to another refactoring.

In a typical usage scenario, a user selects a particular refactoring (e.g., Extract Method in Figure 5(a)) that he/she wants to apply to his/her code. Then, the tool highlights in the graph the selected refactoring and other related refactorings, which can be used before, after, as part of, or instead of the selected refactoring, with the respective nodes colored in yellow, pink, cyan, purple and tan.

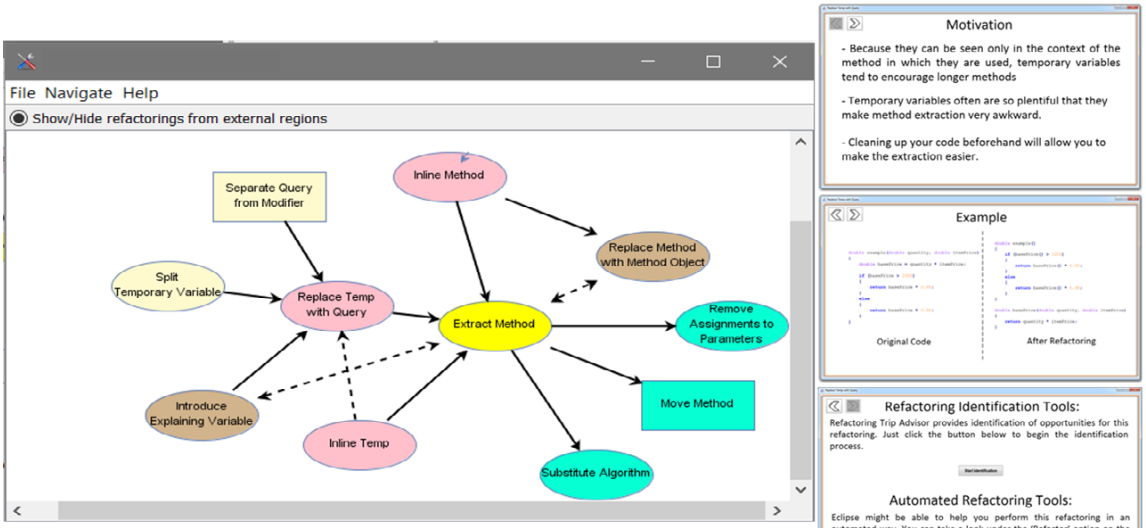
Each refactoring is further related with *slideware* (e.g., Figure 5(b)) that provides guidelines on how to apply it. The slideware consists of three parts: the first part explains the problem solved by the refactoring; the second part, gives a simple example on how to apply the refactoring; the last part, allows the user to execute *a refactoring detector* that identifies refactoring opportunities in the code.

In the initial version of the tool we implemented seven in-house refactoring detectors (Table 7(top)) specifically for the tool, and we reused three third-party detectors (Table 7(bottom)), provided by the JDeodorant[1] refactoring framework. However, the detectors had several design and implementation issues. In particular, the initial version of the tool did not provide any backbone infrastructure for the development of refactoring detectors. Hence, to add new refactoring detectors or existing third-party refactoring detectors we had to re-implement the overall refactoring detection process. When it comes to maintainability and code quality, the refactoring detectors had a significant amount of *duplicate code*, due to the repeated implementation of common refactoring detection steps (scope and subject identification).

To deal with the aforementioned issues we re-engineered the Refactoring Trip Advisor, with respect to the Three-Step Refactoring Detector pattern. The pattern structure facilitates the systematic addition of new in-house refactoring detectors and the reuse of existing third-party refactoring detectors, because the common steps of the refactoring detection process are now encapsulated in the core classes of the pattern structure. Consequently, the classes that implement the refactoring detectors are simpler, smaller and more clean. To incorporate the core functionalities of the JDeodorant refactoring detectors in the pattern structure we developed respective facades, as discussed in the examples (Figure 4).

Indicatively, Table 8 compares the sizes of the refactoring detectors in the initial and the re-engineered versions of the tool, in terms of physical lines of code; as reported by Sjøberg et al. [17], this metric is a good indicator of the effort that is required for code maintenance. Figure 6, further compares the statistical distributions of the sizes in the initial and the re-engineered versions of the tool. Specifically, the figure provides a box-and-whisker plot that depicts the distribution of sizes for the refactoring detectors of each version, in terms of the following statistical values: min, 1st quartile, median, 3rd quartile, max, and average. Clearly, all the statistical

[1] http://users.encs.concordia.ca/~nikolaos/jdeodorant/



(a) Recommended refactorings

(b) Refactoring slides

Figure 5: the Refactoring Trip Advisor overview: (a) Refactoring recommendations on the graph, (b) refactoring slides and opportunity detection.

Table 8: Comparing the sizes of the refactoring detectors in the initial and the re-engineered versions of the Refactoring Trip Advisor.

	Size in physical lines of code (ploc)			
Refactoring Detector	**Initial version**	**Re-engineered version**	**size reduction**	**% of size reduction**
ExtractClassIdentification	307	284	23	7.49%
ExtractMethodIdentification	393	320	73	18.57%
InlineMethodIdentification	286	233	53	18.53%
InlineTempIdentification	420	363	57	13.57%
IntroduceExplainingVariableIdentification	604	529	75	12.41%
IntroduceParameterObjectIdentification	221	167	54	24.43%
MoveMethodIdentification	247	205	42	17.00%
RemoveAssignmentsToParametersIdentification	416	369	47	11.29%
ReplaceMethodWithMethodObjectIdentification	293	235	58	19.79%
ReplaceTempWithQueryIdentification	429	362	67	15.61%
SplitTemporaryVariableIdentification	425	358	67	15.76%
Total	**4041**	**3425**	**616**	**15.24%**

values in the re-engineered version are smaller than the respective values in the initial version of the Refactoring Trip Advisor.

3 CONCLUSION

In this paper, we discussed the key concepts of the Three-Step Refactoring Detector pattern that facilitates the development of extensible and maintainable tools for the detection of refactoring opportunities. The backbone of the pattern is a polymorphic hierarchy of template classes that realize a general three-step refactoring detection process. This structure makes the addition of new refactoring detectors to a tool easy. Moreover, it facilitates the reuse of existing refactoring detectors, provided by third-party developers.

We have used the Three-Step Refactoring Detector pattern to improve the design and implementation of our tool, the Refactoring Trip Advisor. Based on the pattern, we developed a new version of the tool that comprises an extensible set of smaller and cleaner

refactoring detectors that are free from duplicate code. An open issue for further research concerns the assessment of the pattern in a user study that involves third-party developers.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Veli-Pekka Eloranta, for his constructive comments and suggestions during the preparation of this paper.

REFERENCES

[1] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. 2014a. Automating Extract Class Refactoring: An Improved Method and its Evaluation. *Empirical Software Engineering* 19, 6 (2014), 1617–1664.

[2] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. 2014b. Methodbook: Recommending Move Method Refactorings via Relational Topic Models. *IEEE Transactions on Software Engineering* 40, 7 (2014), 671–694.

[3] J. Al Dallal. 2015. Identifying Refactoring Opportunities in Object-Oriented Code: A Systematic Literature Review. *Information and Software Technology* 58, 0 (2015), 231 – 249.

[4] B. Du Bois, S. Demeyer, and J. Verelst. 2004. Refactoring: Improving Coupling and Cohesion of Existing Code. In *Proceedings of the 11th IEEE Working Conference on Reverse Engineering (WCRE)*. 144–151.

[5] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou. 2012. Identification and Application of Extract Class Refactorings in Object-Oriented Systems. *Journal of Systems and Software* 85, 10 (2012), 2241–2260.

[6] M. Fowler. 2000. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley.

[8] M. Kim, T. Zimmermann, and N. Nagappan. 2014. An Empirical Study of Refactoring Challenges and Benefits at Microsoft. *IEEE Transactions on Software Engineering* 40, 7 (2014), 633–649.

[9] S. Kranas, A. V. Zarras, and P. Vassiliadis. 2015. Fitness Workout for Fat Interfaces: Be Slim, Clean, and Flexible. In *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 526–530.

[10] M. Leppänen, S. Mäkinen, S. Lahtinen, O. Sievi-Korte, A. Tuovinen, and T. Männistö. 2015. Refactoring-a Shot in the Dark? *IEEE Software* 32, 6 (2015), 62–70.

[11] T. Mens and T. Tourwé. 2004. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering* 30, 2 (2004), 126–139.

[12] E. R. Murphy-Hill and A. P. Black. 2008. Refactoring Tools: Fitness for Purpose. *IEEE Software* 25, 5 (2008), 38–44.

[13] William F. Opdyke. 1992. *Refactoring Object-Oriented Frameworks.* Ph.D. Dissertation. Univ. of Illinois - Urbana Champaign.

[14] D. Romano, S. Raemaekers, and M. Pinzger. 2014. Refactoring Fat Interfaces Using a Genetic Algorithm. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 351–360.

[15] V. Sales, R. Terra, L. Fernando Miranda, and M. Tulio Valente. 2013. Recommending Move Method Refactorings Using Dependency Sets. In *Proceedigns of the 20th IEEE Working Conference on Reverse Engineering (WCRE)*. 232–241.

[16] D. Silva, R. Terra, and M. Tulio Valente. 2014. Recommending Automated Extract Method Refactorings. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC)*. 146–156.

[17] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå. 2013. Quantifying the Effect of Code Smells on Maintenance Effort. *IEEE Transactions on Software Engineering* 39, 8 (2013), 1144–1156.

[18] T. Tourwé and T. Mens. 2003. Identifying Refactoring Opportunities Using Logic Meta Programming. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR)*. 91–100.

[19] N. Tsantalis and A. Chatzigeorgiou. 2009. Identification of Move Method Refactoring Opportunities. *IEEE Transactions on Software Engineering* 99, 3 (2009), 347–367.

[20] N. Tsantalis and A. Chatzigeorgiou. 2011. Identification of Extract Method Refactoring Opportunities for the Decomposition of Methods. *Journal of Systems and Software* 84, 10 (2011), 1757–1782.

[21] A. V. Zarras, T. Vartziotis, and P. Vassiliadis. 2015. Navigating through the Archipelago of Refactorings. In *Proceedings of the the Joint 23rd ACM SIGSOFT Symposium on the Foundations of Software Engineering and 15th European Software Engineering Conference (FSE/ESEC)*. 922–925.