Recommending Trips in the Archipelago of Refactorings

Theofanis Vartziotis[1], Apostolos V. Zarras[1], Anastasios Tsimakis[1], and Panos Vassiliadis[1]

Department of Computer Science and Engineering, University of Ioannina, Greece
{tvartzio, zarras, pvassil}@cs.uoi.gr, atsimakis@gmail.com

Abstract. The essence of refactoring is to improve source code quality, in a principled, behavior preserving, one step at the time, process. To this end, the developer has to figure out the refactoring steps, while working on a specific source code fragment. To facilitate this task, the documentation that explains each primitive refactoring typically provides guidelines and tips on how to combine it with further refactorings. However, the developer has to cope with many refactorings and lots of guidelines.
To deal with this problem, we propose *a graph-based model that formally specifies refactoring guidelines and tips* in terms of nodes that correspond to refactorings and edges that denote *part-of*, *instead-of* and *succession relations*. We refer to this model as *the Map of the Archipelago of Refactorings* and we use it as the premise of *the Refactoring Trip Advisor*, a refactoring recommendation tool that facilitates the combination of refactorings. A first assessment of the tool in a practical scenario that involves 16 developers and a limited set of refactorings for composing and moving methods brought out positive results that motivate further studies of a larger scale and scope.

Keywords: Refactoring recommendation, Refactoring graph, Refactoring combination

1 Introduction

Refactoring is a basic prerequisite for keeping our source code clean. The basic idea is to improve source code quality, via a series of small behavior-preserving transformations [11, 4].

"The biggest problem with Extract Method is dealing with local variables, and temps are one of the main sources of this issue. When I'm working on a method, I like Replace Temp with Query to get rid of any temporary variables that I can remove. If the temp is used for many things, I use Split Temporary Variable first to make the temp easier to replace."

The previous quote is from Martin Fowler's catalog of refactorings [4]. What is interesting in this quote is that it provides certain guidelines on how to perform the Extract Method refactoring. To make things easier, it suggests to remove temporary variables *before* the method extraction, using Replace Temp with Query. Taking a step further, the quote suggests to use Split Temporary Variable *before* Replace Temp with Query, so as to facilitate the removal of multi-purpose temporary variables.

Observe the next quote, which suggests using Replace Method with Method Object *instead of* Extract Method, in the case of very complex methods.

> *"Sometimes, however, the temporary variables are just too tangled to replace. I need Replace Method with Method Object. This allows me to break up even the most tangled method, at the cost of introducing a new class for the job."*

Moreover, in the following quote we see *part-of* relations, which dictate how to realize Extract Superclass based on more primitive refactorings like Extract Method and Pull Up Method.

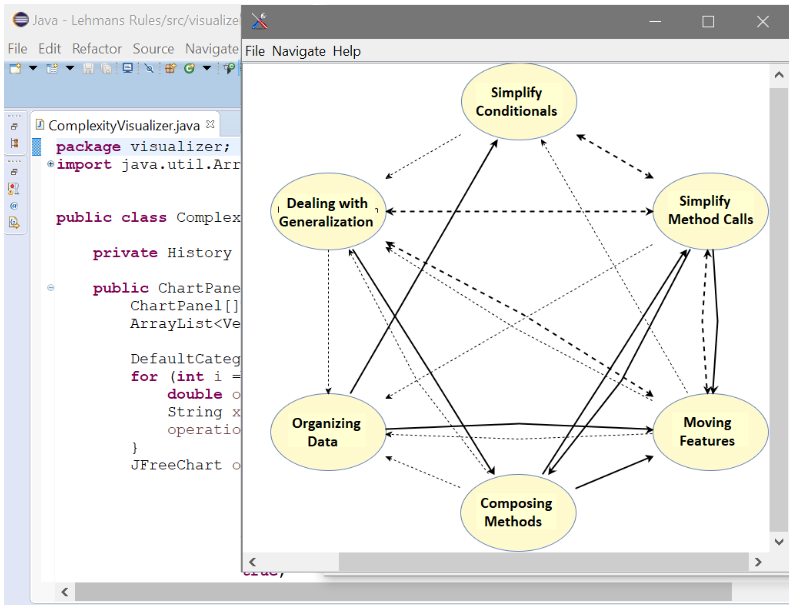
> *"Examine the methods left on the subclasses. See if there are common parts, if there are you can use Extract Method followed by Pull Up Method on the common parts."*

Hence, refactorings come along with several informal guidelines that tell us how to combine them into more complex evolution tasks. *What is the problem with that?* On the one hand, *there are way too many refactorings and guidelines* in Fowler's catalog. Specifically, the catalog consists of 68 different refactorings, while the documentation of these refactorings includes more than 100 guidelines and tips [16]. On the other hand, *the state of the art on refactoring* (two detailed surveys can be found in [6] and [2]) *does not provide means that facilitate the effective exploitation of this knowledge.*

To deal with the aforementioned issues, we propose an approach that allows the developers to combine refactorings into more complex evolution tasks via the following key concepts:

- *The Map of the Archipelago of Refactorings, a graph-based model that specifies informal guidelines and tips, found in Fowler's catalog*, in terms of nodes that correspond to refactorings, and edges that signify *part-of*, *instead-of* and *succession relations* between them. In Zarras et al. [16] we introduced a coarse sketch of the map, while in this paper we provide its detailed formal definition.
- *The Refactoring Trip Advisor*, a refactoring recommendation facility that provides *an interactive perspective of the archipelago map*, which makes suggestions regarding which refactoring(s) to use before, after, or instead of a particular refactoring. The Refactoring Trip Advisor further provides *guidelines on how to apply individual refactorings*, and enables the *identification of refactoring opportunities.*

We assess our approach in two steps: (1) we show that the Refactoring Trip Advisor adheres to the basic refactoring tool principles, recommended by


Java - Lehmans Rules/src/visualize
File Edit Refactor Source Navigate
ComplexityVisualizer.java
File Navigate Help
Simplify Conditionals
Dealing with Generalization
Simplify Method Calls
Organizing Data
Moving Features
Composing Methods

```
package visualizer;
import java.util.Arr

public class Complex

    private History

    public ChartPane
        ChartPanel[]
        ArrayList<Ve

        DefaultCateg
        for (int i =
            double o
            String x
            operatio
        }
        JFreeChart o
```

Fig. 1. The archipelago hyper-map.

Murphy-Hill and Black in [10]; (2) we validate that the Refactoring Trip Advisor can be successfully used in a realistic re-engineering scenario, in a study that involves 16 developers with varying profiles.

The rest of this paper is structured as follows. In Section 2, we discuss related work. In Section 3, we detail the *modus operandi* of the proposed approach. In Section 4, we concentrate on the validation of the proposed approach. Finally, in Section 5 we summarize our contribution and point out the future directions of this work.

2 Related Work

Opdyke introduced refactoring as a behavior preserving process that changes a software, so as to enable other changes to be made more easily [11]. Mens and Tourwé [6] provide an excellent survey that addresses several different aspects of the refactoring process (e.g., refactoring activities, techniques, supporting tools). A more recent extensive survey that focuses on techniques and tools for the detection of refactoring opportunities is provided by Al Dallal [2].

An important result that is brought out by the empirical study of Kim et al. [5] is the need to combine refactorings in more complex evolution tasks. To deal with this issue, the state of the art comprises a number of interesting search-based refactoring approaches (e.g., [9]) that apply refactorings automatically towards maximizing the software quality improvement, with respect to a set of target quality indicators. Our approach follows a different direction, as the goal is to widen the developer's choices with recommendations derived from the proposed graph-based refactoring model.

Our work is more closely related to approaches that concern the modeling of refactoring relations. In particular, Mens et al. [7] model refactoring relations to detect conflicts between refactorings. Another interesting approach that employs refactoring relations to enable automated refactoring scheduling and conflict resolution is proposed by Moghadam and Cinnéide [8]. Van Der Straeten et al [15], rely on refactoring relations to preserve program behavior. The key difference of our approach from these efforts is that we formally model guidelines and tips found in Fowler's catalog of refactorings [4], in terms of an interactive model that provides actionable recommendations for the effective combination of refactorings.

When it comes to the detection of refactoring opportunities [2], the goal of our approach is to facilitate the integration of different existing techniques under the common umbrella of the proposed graph-based refactoring model. As a proof of concept, we have done this in the Refactoring Trip Advisor with three different refactoring detection techniques [12, 13, 3] that are provided by the JDeodorant framework.

3 Refactoring Trip Advisor

In this section, we formally model the map of the archipelago of refactorings. Then, we focus on the recommendation of refactoring trips. Finally, we illustrate the role of our approach in a realistic re-engineering scenario.

3.1 Modelling Refactoring Relations

At a glance, *the Map of the Archipelago of Refactorings* models informal guidelines and tips in terms of different relations between refactorings. Our baseline is Martin Fowler's catalog of refactorings [4]. Nevertheless, the extension of the map with further refactorings and relations is straightforward. The core concept of the map is a graph, with nodes representing refactorings and edges representing the relations between them. As relations are of different kinds, we introduce corresponding types of edges. More formally, we define the overall model as follows.

Definition 1. ***[Archipelago Map]*** *The map of the archipelago of refactorings is a directed graph $\mathbb{M}_{\mathbb{R}}(\mathbb{V}_{\mathbb{R}}, \mathbb{E}_{\mathbb{R}})$, s.t. the nodes $\mathbb{V}_{\mathbb{R}}$ represent refactorings, while the edges $\mathbb{E}_{\mathbb{R}}$ denote relations that correspond to guidelines and tips, concerning the combination of refactorings:*

- ***[Node Properties]*** $\mathbb{V}_{\mathbb{R}} = \bigcup_{i=1}^{6} V_R^i$ *is divided into disjoint subsets, called regions. The regions correspond to the different categories of refactorings (Figure 1), defined in Fowler's catalog [4].*

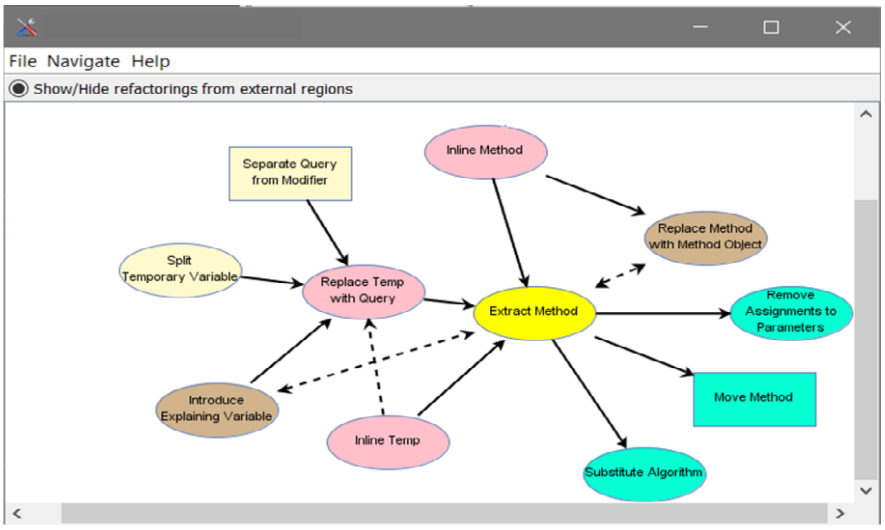

File Navigate Help
Show/Hide refactorings from external regions
Separate Query from Modifier
Inline Method
Replace Method with Method Object
Split Temporary Variable
Replace Temp with Query
Extract Method
Remove Assignments to Parameters
Move Method
Introduce Explaining Variable
Inline Temp
Substitute Algorithm

Fig. 2. The *Composing Methods* region map.

– *[**Edge Properties**] The edges $e(v_i, v_j) \in \mathbb{E}_{\mathbb{R}}$ are typed, with $type(e(v_i, v_j)) \in \mathbb{T} = \{succession, part\ of, instead\ of\}$:*
  - *A <u>succession</u> relation, is represented as a solid unidirectional edge; it denotes that it would be useful to perform the source (resp. target) refactoring v_i (resp. v_j), before (resp. after) the target (resp. source) refactoring v_j (resp. v_i).*
  - *A <u>part of</u> relation is represented as a dotted unidirectional edge between two refactorings; it signifies that the source refactoring v_i, can be used for the realization of the target refactoring v_j.*
  - *A <u>instead of</u> relation, is denoted as a dashed bidirectional edge; it means that either one of the related refactorings can be used, instead of the other.*

The archipelago map is complex, consisting of 68 nodes and 101 edges between them [1]. The complexity of the map further points out the amount of information and the effort that is required from the developer, to exploit refactorings in an effective way. To deal with the complexity of the map, we decompose it into a set of *region maps* (e.g., Composing methods region given in Figure 2). A region map shows the refactorings of a respective region, along with important refactorings from other regions that are related to them.

Definition 2. *[**Region Map**] For every region V_R^i of $\mathbb{V}_{\mathbb{R}}$, the region map $M_{RG}^i(V_{RG}^i, E_{RG}^i)$ is an induced sub-graph of $\mathbb{M}_{\mathbb{R}}$, s.t. V_{RG}^i consists of the refactorings V_R^i and related refactorings from other regions, i.e., $(V_{RG}^i \supset V_R^i) \wedge (\forall v_j \in V_{RG}^i - V_R^i,\ \exists v_i \in V_R^i\ \ s.t.\ \ (e(v_i, v_j) \in \mathbb{E}_{\mathbb{R}} \vee e(v_j, v_i) \in \mathbb{E}_{\mathbb{R}}))$.*

[1] The map can be found at: www.cs.uoi.gr/~zarras/RefactoringsArchipelagoWEB/ArchipelagoOfRefactorings.html

At a higher-level of abstraction the region maps are organized with respect to *a hyper-map* (Figure 1). In a sense, the archipelago hyper-map provides a summary of the full-fledged archipelago map.

Definition 3. ***[Archipelago Hyper-map]*** *The archipelago hyper-map* $\mathbb{H}_{\mathbb{R}}(\mathbb{V}_{\mathbb{H}}, \mathbb{E}_{\mathbb{H}})$*, is a directed graph, s.t. the nodes of the graph represent the regions* $V_R^1, V_R^1, \ldots V_R^6$ *of the archipelago map and the edges of the hyper-map are produced with respect to the archipelago map* $\mathbb{M}_{\mathbb{R}}(\mathbb{V}_{\mathbb{R}}, \mathbb{E}_{\mathbb{R}})$*, as follows:* $\forall e(v_i, v_j) \in \mathbb{E}_{\mathbb{R}}, \quad \exists e(V_R^i, V_R^j) \in \mathbb{E}_{\mathbb{H}} \;\; if \; and \; only \; if \;\; (v_i \in V_R^i \wedge v_j \in V_R^j \wedge V_R^i \neq V_R^j)$.

3.2 Recommending Refactoring Trips

To facilitate the refactoring process, *the archipelago map must go live to provide actionable recommendations* to the developer towards the effective combination of refactorings. Moreover, the developer needs *contextualization* for each refactoring, in the form of guidelines concerning how to apply it. Finally, the developer needs assistance for the *identification of refactoring opportunities* in the specific module (package, class, method, etc.) that he/she is working with.

To deal with the aforementioned issues we developed *the Refactoring Trip Advisor* as an Eclipse plugin [2]. At a glance, *a refactoring trip* begins with the developer selecting a particular refactoring region from the archipelago hyper-map (e.g., the Composing method region given in Figure 1). As a result, the Refactoring Trip Advisor provides to the developer the map of the selected region. The developer selects the particular refactoring (e.g., Extract Method in Figure 2) that he/she wants to apply. Then, the Refactoring Trip Advisor highlights in the map 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 related with *slideware* 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 developer to execute available *refactoring detectors* that identify refactoring opportunities in the code. To perform the recommended refactorings the developer can exploit the available IDE refactoring facilities (see the related discussion in Section 4.1).

Regarding the refactoring detectors, one of the primary concerns of our approach is *extensibility*. Specifically, our goal is to ease the integration of the tool with (a) *in-house refactoring detectors*, developed specifically for our approach and (b) *external refactoring detectors*, provided by third-party developers. To achieve this goal, we rely on the Three-Steps Refactoring Detector pattern that we introduced in Tsimakis et al. [14]. The pattern facilitates the development of refactoring detectors via a polymorphic hierarchy of template classes that realize a general three-step refactoring detection process. In the Refactoring Trip Advisor we used the pattern for the development of eight in-house refactoring detectors. Moreover, we used the pattern to facilitate the integration of

[2] The plugin source code can be downloaded from github.com/AnastasiosHJW/RefactoringTripAdvisor

the Refactoring Trip Advisor with three external refactoring detectors that are provided by the JDeodorant refactoring framework [12, 13, 3]. More details concerning the usage of the pattern in the Refactoring Trip Advisor can be found in Tsimakis et al. [14].

To discuss the involvement of the Refactoring Trip Advisor in the refactoring process we employ a typical re-engineering scenario that concerns data containers and misplaced responsibilities. Specifically, we focus on an application that analyzes the evolution of Amazon Web Services (AWS) [17]. The heart of the application is the `History` class, a data container that keeps the evolution data of subsequent Web service versions. Around the `History` class there are several client classes that manipulate the evolution data. The code of the client methods is typically long and complicated. Certain parts of the client methods are misplaced, as the `History` class should have been responsible for the manipulation of the evolution data.

```java
public ChartPanel[] visualizeGrowth() {
  ChartPanel[] panels = new ChartPanel[2];
  ArrayList<VersionInfo> versionsList = history.getVersions();

  DefaultCategoryDataset operationsGrowth =new DefaultCategoryDataset();
  for (int i = 0; i < versionsList.size(); i++) {
    double opers = versionsList.get(i).getOperationGrowth();
    String xAxis = versionsList.get(i).getId();
    operationsGrowth.setValue(opers, "Operations", xAxis);
  }

  JFreeChart opersGrowthChart = ChartFactory.createLineChart(
    "Growth␣Rate␣Line␣Chart", "Version␣ID",
    "Growth␣Rate", operationsGrowth,
    true, true, false
  );
  panels[0] = new ChartPanel(opersGrowthChart);

  //...............
  return panels;
}
```

Listing 1. Code snippet from `visualizeGrowth()`, before refactoring.

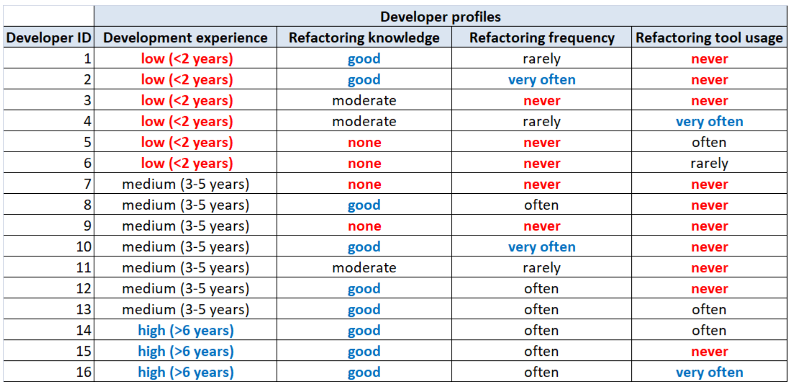
```java
public ChartPanel[] visualizeGrowth() {
  ChartPanel[] panels = new ChartPanel[2];

  DefaultCategoryDataset operationsGrowth = history.getOpersGrowth();

  JFreeChart opersGrowthChart = ChartFactory.createLineChart(
    "Growth␣Rate␣Line␣Chart",
    "Version␣ID", "Growth␣Rate",
    operationsGrowth, true, true, false
  );
  panels[0] = new ChartPanel(opersGrowthChart);

  //...............
  return panels;
}
```

Listing 2. Code snippet from `visualizeGrowth()`, after refactoring.

For instance, `GrowthVisualizer` is a client class that visualizes the Web services' growth, in terms of the number of provided operations and data structures. Listing 1 gives a code snippet from the `visualizeGrowth()` method of

`GrowthVisualizer`. Our goal is to make this snippet smaller and simpler. To this end, we begin from the archipelago hyper-map (Figure 1), which gives general recommendations on how to combine refactorings from different regions. To reorganize the code of the method, we can use refactorings from the Composing Methods region. These refactorings typically result in the extraction of new methods. Therefore, the hyper-map suggests that afterwards it may be useful to consider refactorings from the Moving Features region, to potentially move extracted methods and related fields.

Table 1. Developers profiles (colours highlight boundary values).



	Developer profiles			
Developer ID	Development experience	Refactoring knowledge	Refactoring frequency	Refactoring tool usage
1	low (<2 years)	good	rarely	never
2	low (<2 years)	good	very often	never
3	low (<2 years)	moderate	never	never
4	low (<2 years)	moderate	rarely	very often
5	low (<2 years)	none	never	often
6	low (<2 years)	none	never	rarely
7	medium (3-5 years)	none	never	never
8	medium (3-5 years)	good	often	never
9	medium (3-5 years)	none	never	never
10	medium (3-5 years)	good	very often	never
11	medium (3-5 years)	moderate	rarely	never
12	medium (3-5 years)	good	often	never
13	medium (3-5 years)	good	often	often
14	high (>6 years)	good	often	often
15	high (>6 years)	good	often	never
16	high (>6 years)	good	often	very often

Suppose that we select Extract Method (yellow node in Figure 2) from the Composing Methods region. The respective refactoring detector [13] suggests to extract a new method for the loop (Listing 1, lines 6-10) that prepares the evolution data-sets for the visualization of the Web service growth. Before the method extraction, the Composing Methods map recommends the use of refactorings like Inline Temp and Replace Temp with Query (pink nodes in Figure 2) for the removal of local variables. These refactorings shall make the method extraction easier (see first quote in Section 1). The Inline Temp detector [14] identifies several such variables (e.g., `versionList`, `opers`, `xAxis`) After the method extraction, the map suggests to consider Move Method (cyan node in Figure 2) to place the extracted methods (`getOpesGrowth()`) close to the data that it manipulates, while the Move Method detector [12] identifies `History` as the appropriate target class. Overall, the refactored code snippet is given in Listing 2.

4 Validation

To validate the Refactoring Trip Advisor we consider two main issues. First, we assess whether it adheres to the basic refactoring tool principles, introduced by Murphy-Hill and Black [10]. Then, we examine what do the developers actually think about the tool.

4.1 Fitness for purpose

Murphy-Hill and Black introduced five principles for refactoring tools [10]. Specifically, a refactoring tool should allow the programmer to: (**R1**) Choose the desired refactoring quickly; (**R2**) switch seamlessly between program editing and refactoring; (**R3**) view and navigate the program code while using the tool; (**R4**) avoid providing explicit configuration information; (**R5**) access all the other tools normally while using the tool.

The Refactoring Trip Advisor facilitates the selection of refactorings (**R1**) based on two concepts: (1) it provides the region maps that hide from the developer the complexity of the entire archipelago map; (2) it provides the archipelago hyper-map that summarizes the contents of the archipelago map. The basic functionalities of the tool are provided via a separate frame (Figures 1, 2). The developer can activate this frame and switch back to the main IDE frame, at any point, to edit/view/navigate the program code (**R2**, **R3**), and use any other tool that is available through the IDE (**R5**). The recommendation of refactorings based on the region maps does not require any configuration information. On the other hand, certain detectors of refactoring opportunities do. Typically, the required information concerns thresholds that customize the detectors' modus operandi. Nevertheless, the developer can avoid setting these thresholds (**R4**), as the proposed tool assumes default values for them.

4.2 The developers' opinions

The *goal* of this study is to let the developers *use* the Refactoring Trip Advisor and get their *feedback* concerning the overall approach. To familiarize the developers with the Refactoring Trip Advisor we employed the re-engineering scenario introduced in Section 3. Specifically, we asked the developers to simplify the `visualizeGrowth()` method (a code snippet of the method is given in Listing 1) with the help of the tool. As a starting point, we prompted the developers to focus on refactorings from the Composing Methods region. After this experience, we asked them to perform an overall review of the the Refactoring Trip Advisor, by rating the usefulness of the refactoring relations, slideware, and detection facilities, based on a typical 5-level likert scale. The developers could provide further comments, remarks, and suggestions, concerning our approach.

Our study involves a sample of 16 developers from industry and academia (students and staff members). The selection was based on purposive sampling of heterogeneous instances [1]; the developers were chosen deliberately to reflect diversity on (a) development experience, (b) knowledge about refactoring, (c) frequency of refactoring, and (d) usage of refactoring tools. The detailed profiles of the developers are provided in Table 1.

Key findings. Table 2 summarizes the results that we obtained. The results are provided in three parts: the first part (Table 2 - left) gives the *statistical breakdown of the refactorings* that have been performed by the developers; the second part (Table 2 - middle) analyzes the *exploitation of the recommendations* that have been provided by the Refactoring Trip Advisor, in terms of the percentage

Table 2. Refactorings & tool feature ratings (colours highlight boundary values).

	Refactorings performed				Exploitation of recommendations		Tool features rating		
Developer ID	Inline Temp	Extract Method	Move Method	Total	% of suggested refactorings that have been perfomed	% of performed refactorings that have been suggested	Relations	Slides	Detectors
1	7	2	2	11	100,00%	100,00%	5	5	5
2	6	2	0	8	72,73%	100,00%	4	5	4
3	2	0	0	2	18,18%	100,00%	5	5	5
4	7	2	2	11	100,00%	100,00%	5	5	4
5	3	2	0	5	45,45%	100,00%	4	5	3
6	3	2	2	7	63,64%	100,00%	4	4	2
7	4	1	0	5	45,45%	100,00%	4	5	3
8	7	2	0	9	81,82%	100,00%	5	5	5
9	7	0	0	7	63,64%	100,00%	5	4	4
10	7	3	2	12	100,00%	91,67%	4	4	4
11	7	2	2	11	100,00%	100,00%	5	5	5
12	7	2	2	11	100,00%	100,00%	5	5	5
13	0	2	2	4	36,36%	100,00%	4	4	4
14	7	2	0	9	81,82%	100,00%	4	5	3
15	7	2	2	11	100,00%	100,00%	4	4	3
16	5	2	2	9	81,82%	100,00%	3	3	3
Average	5,38	1,75	1,13	8,25	74,43%	99,48%	4,38	4,56	3,88
Stdev	2,28	0,77	1,02	3,00	26,60%	2,08%	0,62	0,63	0,96

of recommended refactorings that have been performed by the developers, and the percentage of performed refactorings that have been recommended; finally, the third part (Table 2 - right) details the assessment of the tool's features.

Overall, *the developers managed to use the Refactoring Trip Advisor in the context of a realistic task*. Nevertheless, *the exploitation of the provided recommendations by the developers varies, along with the quality of the results that they produced*. In particular, two developers (3 and 9) just removed local variables from `visualizeGrowth()`, via Inline Temp. Five developers (2, 5, 7, 8, 9) simplified `visualizeGrowth()` even more, using Extract Method. Finally, *nine developers* (1, 4, 6, 10, 11, 12, 13, 15, 16) *solved the problem of misplaced responsibilities*, by moving the extracted methods to the `History` class, via Move Method.

The percentage of recommended refactorings that have been performed by the developers varies from 18.18% to 100%. However, *most of the developers applied a large percentage of the recommended refactorings*; twelve developers performed more than 60% of the recommended refactorings, while the average percentage of refactorings that have been performed is 74.43% with a standard deviation of 26.60%. On the other hand, *the percentage of performed refactorings that have been recommended is 100% for all the developers, except one*, who extracted a method that was not included in the suggestions.

Concerning the assessment of the overall approach, *the developers provided quite high ratings, with the refactoring relations and slides being the most appreciated features*, followed by the refactoring detection facilities. Regarding further comments and remarks, several developers pointed out that the proposed approach *helped them to learn more about refactoring*. Moreover, they mentioned that the proposed approach could be considered both for *development* and *education* purposes. The developers' *suggestions* ranged from *concrete improvements* (e.g., to reduce the number of pop-up windows and make the slideware resizable), to *broader ideas* like making the representation of the map more interactive, allowing the developer to customize it by adding/removing/changing refactorings and relations, adding more slides on the relations between refactorings, complementing the slideware with audio/video, and so on.

Threats to Validity. Two factors that threaten the internal validity of one group experiments are *history* and *maturation*; the longer the time of the experiment, the more likely are these threats [1]. In our study, the overall duration of the tests was reasonably short, ranging from 15 to 65 mins. Another threat to internal validity is the social desirability bias, in the sense that the developers simply agreed with the tool recommendations. To deal with this threat, we used anonymous questionnaires. Our assessment relies on a single scenario that focuses on a subset of refactorings for composing and moving methods. These are threats to external validity. We further asked from the developers an overall review, concerning the proposed approach as a whole. Nevertheless, to be able to generalize the results further studies should be performed, involving more developers, subject systems and refactorings.

5 Conclusion

In this paper, we proposed an approach that provides actionable recommendations for the effective combination of refactorings. The recommendations are based on an interactive model that formally specifies respective informal refactoring guidelines. The proposed approach is inline with the fundamental refactoring tool principles. To further assess the approach, we conducted a study that involved 16 developers and a limited set of refactorings for composing and movings methods. The developers found the overall approach useful. The positive results that we obtained encourage follow up studies of a broader scale and scope.

Our approach is currently based on knowledge that is "hidden" in Fowler's catalog. Other sources of information, views and notations can be considered towards its extension. Making the refactoring detection techniques more developer-intuitive and easy to use is also an issue that should be further investigated. In the future, it would also be interesting to investigate the relation between the proposed approach and the issue of technical debt prioritization.

Acknowledgements We would like to thank the anonymous reviewers for their feedback on the paper.

References

1. Cook, T.D., Campbell, D.T.: Quasi-Experimentation: Design and Analysis Issues for Field Settings. Houghton Mifflin Company (1979)

2. Dallal, J.A.: Identifying Refactoring Opportunities in Object-Oriented Code: A Systematic Literature Review. Information and Software Technology 58(0), 231 – 249 (2015)
3. Fokaefs, M., Tsantalis, N., Stroulia, E., Chatzigeorgiou, A.: Identification and Application of Extract Class Refactorings in Object-Oriented Systems. Journal of Systems and Software 85(10), 2241–2260 (2012)
4. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (2000)
5. Kim, M., Zimmermann, T., Nagappan, N.: An Empirical Study of Refactoring Challenges and Benefits at Microsoft. IEEE Transactions on Software Engineering 40(7), 633–649 (2014)
6. Mens, T., Tourwé, T.: A Survey of Software Refactoring. IEEE Transactions on Software Engineering 30(2), 126–139 (2004)
7. Mens, T., Taentzer, G., Runge, O.: Analysing Refactoring Dependencies Using Graph Transformation. Software and System Modeling 6(3), 269–285 (2007)
8. Moghadam, I.H., Cinnéide, M.Ó.: Resolving Conflict and Dependency in Refactoring to a Desired Design. e-Informatica 9(1), 37–56 (2015)
9. Morales, R., Chicano, F., Khomh, F., Antoniol, G.: Efficient Refactoring Scheduling Based on Partial Order Reduction. Journal of Systems and Software 145, 25–51 (2018)
10. Murphy-Hill, E.R., Black, A.P.: Refactoring Tools: Fitness for Purpose. IEEE Software 25(5), 38–44 (2008)
11. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. Ph.D. thesis, Univ. of Illinois - Urbana Champaign (1992)
12. Tsantalis, N., Chatzigeorgiou, A.: Identification of Move Method Refactoring Opportunities. IEEE Transactions on Software Engineering 99(3), 347–367 (2009)
13. Tsantalis, N., Chatzigeorgiou, A.: Identification of Extract Method Refactoring Opportunities for the Decomposition of Methods. Journal of Systems and Software 84(10), 1757–1782 (2011)
14. Tsimakis, A., Zarras, A.V., Vassiliadis, P.: The Three-Step Refactoring Detector Pattern. In: Proceedings of the 24th European Conference on Pattern Languages of Programs (EuroPLoP). p. to appear (2019), www.cs.uoi.gr/~zarras/papers/C36.pdf
15. Van Der Straeten, R., Jonckers, V., Mens, T.: A Formal Approach to Model Refactoring and Model Refinement. Software and System Modeling 6(2), 139–162 (2007)
16. Zarras, A.V., Vartziotis, T., Vassiliadis, P.: 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). pp. 922–925 (2015)
17. Zarras, A.V., Vassiliadis, P., Dinos, I.: Keep Calm and Wait for the Spike! Insights on the Evolution of Amazon Services. In: Proceedings of the 28th International Conference on Advanced Information Systems Engineering (CAiSE). pp. 444–458 (2016)