Navigating through the Archipelago of Refactorings

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

Theofanis Vartziotis
Department of Computer Science & Engineering
University of Ioannina
Greece
tvartzio@cs.uoi.gr

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

ABSTRACT

The essence of refactoring is to improve software quality via the systematic combination of primitive refactorings. Yet, there are way too many refactorings. Choosing which refactorings to use, how to combine them and how to integrate them in more complex evolution tasks is really hard. Our vision is to provide the developer with *a "trip advisor" for the archipelago of refactorings*. The core idea of our approach is *the map of the archipelago of refactorings*, which identifies the basic relations that guide the systematic and effective combination of refactorings. Based on the map, the trip advisor makes suggestions that allow the developer to decide how to start, assess the possible alternatives, have a clear picture of what has to be done before, during and after the refactorings and assess the possible implications.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-Oriented Programming*

Keywords

Refactoring map, Refactoring composition, Refactoring recommendation

1. INTRODUCTION

> *Refactoring is risky ... You start digging in the code. Soon you discover new opportunities for change, and you dig deeper. The more you dig, the more stuff you turn up ... and the more changes you make. Eventually you dig yourself into a hole you can't get out of. To avoid digging your own grave, refactoring must be done systematically ... Erich Gamma*

Problem. The above quote, taken from E. Gamma's forward in M. Fowler's book [4], highlights the importance of refactoring in a systematic way. To this end, in [4], Fowler (with contributions by Beck, Brand, Opdyke and Roberts) introduced a well known catalog of 72 refactorings. Yet, developers -even developers of refactoring tools- are not inclined to using tools to automate the refactoring of their code [8, 6], *as choosing which refactorings to use, how to combine them and how to integrate them in more complex evolution tasks is really hard*. The need of systematic usage and the management of a very large space of options are reasons that make the task hard. To address the vastness and complexity of the refactorings space, Fowler [4] had already grouped the 72 refactorings in six groups (see also Section 2 for details). Even so, exactly due to the inherent interconnection of these refactorings, the space of options before, during and after a refactoring is still large. Specifically, in Fowler's catalog, the documentation of each refactoring includes discussions that reflect relations with other refactorings which could be combined with the target refactoring, towards having more effective results. The "hidden" relations, refer to refactorings that could be performed before, or after the target refactoring. The "hidden" relations also concern alternative refactorings that could be performed instead of the target refactoring, or constituent refactorings that can be used to realize the target refactoring. Overall, in Fowler's catalog there are more than 100 implicit relations between refactorings (see Sec. 2 for details).

We believe that refactoring should be addressed with a fresh look, compared to the existing landmark view of [4]. We should accept that refactorings, in practice, are inherently more complex than what Fowler said. We should also accept that when dealing with refactorings, the developer is not facing just *one* refactoring in isolation, but rather, a composition of them, with unclear outcomes, side-effects and alternatives.

State of the art. Currently, although refactoring is an active research area, the state of the art is simply not aligned with the aforementioned concerns. The focus of the state of the art is on techniques for (semi)automated refactoring. Certain techniques consider a single refactoring (e.g., [3, 1]), while certain others provide support for a limited subset of refactorings (e.g., [5, 9]). Various data mining (e.g., [3]) and optimization techniques (e.g., [5]) have been employed to automate refactoring. According to [2] the state of the art provides automated support only for 27% of the refactorings that are given in Fowler's catalog. The interested reader may refer to [7] and [2] for two detailed surveys on refactoring.

Vision. *Our vision is to arm the developer with a "trip advisor" for the vast space of refactorings that will allow the*

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2803203

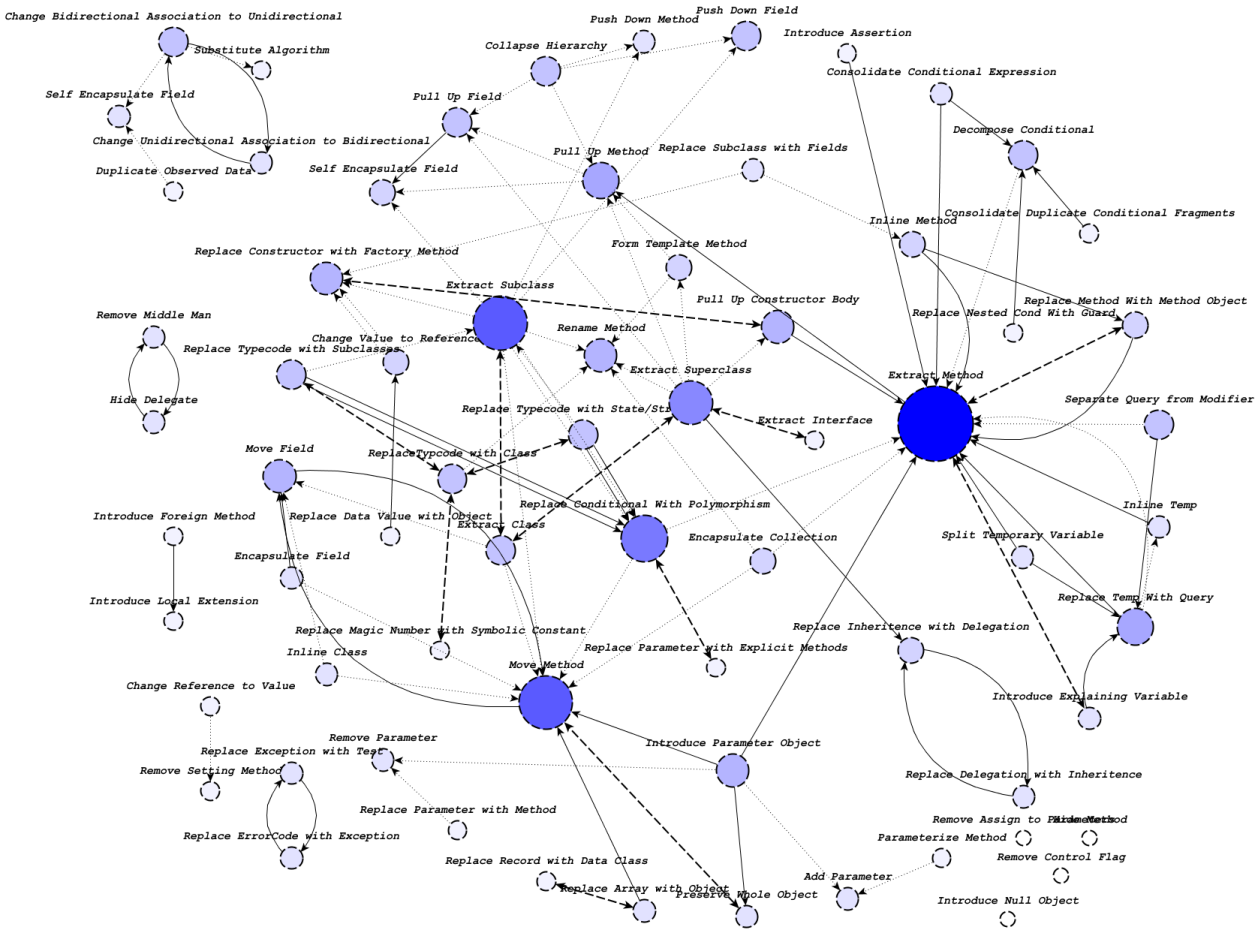

Figure 1: The full-fledged map of the archipelago of refactorings.

developer (a) to decide how to start, (b) assess the possible alternatives and combine refactorings effectively, (c) have a clear picture of what has to be done before, during and after the refactorings and assess the possible implications.

2. APPROACH

Overall, our study of Fowler's catalog brought out more than 100 relations between refactorings. In order to represent them concretely, we have introduced the *map of the archipelago of refactorings*, which is a graph with nodes representing Fowler's refactorings and edges representing their relations (details coming right away on edges). In Figure 1, we give a full-fledged view of the map. The size of the nodes and the tone of their colour indicate their importance in the map, concerning the total amount of relations (i.e., the sum of the node fan-in and fan-out) they are involved in.

The archipelago map is a cornerstone of our approach, as *the refactoring trip advisor adapts the map to the developers' context and helps them navigate through the archipelago of refactorings, by providing practical advice concerning the systematic and effective combination of refactorings.*

What is the map of the archipelago of refactorings? In detail, the map is a graph that consists of 72 nodes, one for each refactoring included in Fowler's catalog. The in-depth study of Fowler's catalog, and specifically the mechanics of each refactoring, revealed 3 different kinds of relations between refactorings, which correspond to different types of edges between the nodes of the map:

- *Succession*: A *succession relation*, represented as a solid unidirectional edge between two refactorings, signifies that it would be useful for the developer to perform the source refactoring, before the target refactoring. Equivalently, it also means that it would be useful to perform the target refactoring, after the source refactoring.

- *Part-of*: A *part-of relation*, denoted as a dotted unidirectional edge between two refactorings, means that the developer could employ the target refactoring for the realization of the source refactoring.

- *Instead-of*: A *instead-of relation*, represented as a dashed bidirectional arrow between two refactorings, means that depending on the circumstances the developer can consider using either one of the related refactorings, instead of the other.

Clearly, the archipelago map is a very complex graph (its name is, of course, indicative of its nature). The complexity of the graph highlights the amount of information and the

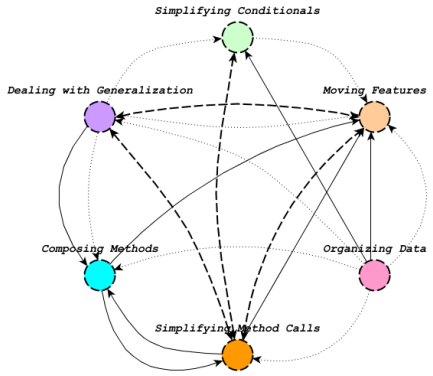

Figure 2: Zoom out: A usable view of the archipelago map at the region level.

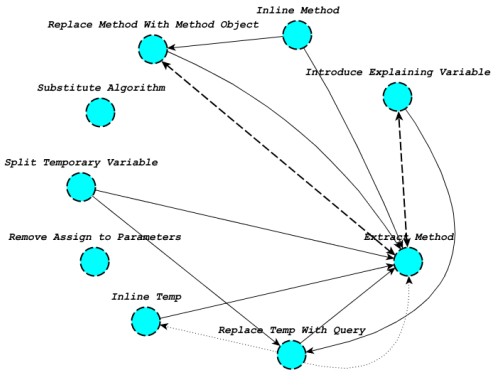

Figure 3: Zoom in: Composing Methods region.

mental effort that is required from the developers, for the effective combination of refactorings.

Thus, to ease navigation, we divide the map in 6 major subgraphs, which we call *regions*. The regions correspond to the 6 different categories of refactorings that are identified in Fowler's catalog: *Composing Methods* consists of refactorings that allow to package code properly in methods; *Moving Features* concerns refactorings that focus on responsibility assignment; *Organizing Data* consists of refactorings that make working with data easier; *Simplifying Conditionals* offers refactorings that deal with the complexity of conditional logic; *Simplifying Method calls* provides refactorings that make method prototypes easier to understand and use; *Dealing with Generalization* consists of refactorings that improve class hierarchies.

In Figure 2, we give a zoom-out of the map that depicts the region-level organization of refactorings. Succession, part of, and instead of relations between regions signify that corresponding relations exist between the refactorings that are included in these regions. On the other hand, Figure 3 zooms into the Composing Methods region.

How to navigate through the archipelago of refactorings? Using the map of the archipelago of refactorings involves adapting it into a specific context, i.e., a piece of code to be refactored. For instance, suppose you are about to reorganize the code of the `visualizeEvolutionHistoryData()` method that is given in Listing 1. The method visualizes software evolution history data. The code of the method is long and complex. To deal with this problem you plan to use the Extract Method refactoring, from the Composing Methods region. The relations at the level of the Composing Methods region (Figure 3) tell you that the extraction will be easier if before that you try to reduce the amount of local variables used in the code, via refactorings like Inline Temp and Replace Temp with Query. Less local variables will result in fewer parameters for the methods that will be extracted [4]. However, even before that, the map says that you should try to clean up local variables that serve more than one purpose by employing the Split Temporary Variable refactoring. Nevertheless, if the method extraction is too difficult due to the high number of variables and the messy code, the map suggests using Replace Method with Method Object, instead of Extract Method. In our example, Inline Temp can be applied in the case of the `versionsList` variable, while Replace Temp with Query makes sense for `startDate`, `endDate`, `lifetimeDuration`, and `year`.

```
1 public ChartPanel visualizeEvolutionHistoryData(){
2
3     ArrayList<VersionInfo> versionsList = history.getVersions();
4         DefaultCategoryDataset objDataset = new DefaultCategoryDataset();
5         int[][] versionsPerYear;
6         int startDate= versionsList.get(0).yearOfDate();
7         int endDate = versionsList.get(versionsList.size()-1).yearOfDate();
8         int lifetimeDuration = endDate-startDate+1;
9         versionsPerYear = new int[lifetimeDuration][2];
10
11        for(int i=0; i < lifetimeDuration; i++){
12                versionsPerYear[i][0] = startDate + i;
13        }
14
15        for(int i=0; i < versionsList.size(); i++){
16                int year = versionsList.get(i).yearOfDate();
17                for(int j=0; j < lifetimeDuration; j++){
18                        if(versionsPerYear[j][0] == year){
19                                versionsPerYear[j][1]++; break;
20                        }
21                }
22        }
23
24        for(int i=0; i < lifetimeDuration; i++){
25                String xAxisLabel = "'"+(versionsPerYear[i][0]-2000);
26                objDataset2.setValue(versionsPerYear[i][1], "Year Versions"
                          , xAxisLabel);
27        }
28
29        JFreeChart objChart = ChartFactory.createBarChart(
30                "Versions per Year Bar Chart", "Version ID", "Versions",
31                objDataset, PlotOrientation.VERTICAL,
32                true, true, false);
33
34    return new ChartPanel(objChart);
35 }
```

Listing 1: Running example.

Listing 2 gives the outcome of the whole refactoring process. Replace Temp with Query, results in 4 new methods, `getYearOfFirstVersion()`, `getYearOfLastVersion()`, `getLifetimeDuration()`, and `getYearOfRelease()`. Then, the first and the second loop of the original code (Listing 1) are extracted in `initializeVersionsPerYear()` and `populateVersionsPerYear()`, while the rest of the code is extracted in `createVersionsPerYearDataSet()`.

The relations at the region-level (Figure 2) tell you how to combine different categories of refactorings. For instance, the refactoring of the Composing Methods region can be combined with refactorings from the Moving Features region. More specifically, the map says that after the extraction of new methods you should check whether certain of them (possibly along with other features) could be moved across classes. In our example, all the methods that have been extracted manipulate evolution history data; hence, it makes sense to move these methods to the `History` class.

```
1 public ChartPanel visualizeEvolutionHistoryData(){
2     int[][] versionsPerYear = initializeVersionsPerYear();
3     populateVersionsPerYear(versionsPerYear);
4     return createVersionsPerYearDataSet(versionsPerYear);
5 }
6
7 private int[][] initializeVersionsPerYear() {
8     int[][] versionsPerYear = new int[getLifetimeDuration()][2];
9     for(int i=0; i < getLifetimeDuration(); i++){
10      versionsPerYear[i][0] = getYearOfFirstVersion() + i;
11    }
12    return versionsPerYear;
13 }
14
15 private void populateVersionsPerYear(int[][] versionsPerYear) {
16    for(int i=0; i < history.getVersions().size(); i++){
17        for(int j=0; j < getLifetimeDuration(); j++){
18            if(versionsPerYear[j][0] == getYearOfRelease()){
19                versionsPerYear[j][1]++; break;
20            }
21        }
22    }
23 }
24
25 private int getLifetimeDuration() {
26   return getYearOfLastVersion()-getYearOfFirstVersion()+1;
27 }
28
29 private int getYearOfLastVersion() {
30   return history.getVersions().get(
31            history.getVersions().size()-1).yearOfDate();
32 }
33
34 private int getYearOfFirstVersion() {
35   return history.getVersions().get(0).yearOfDate();
36 }
37
38 private int getYearOfRelease() {
39   return history.getVersions().get(i).yearOfDate();
40 }
```

Listing 2: Running example - Refactored.

3. STATUS & NEXT STEPS

Up to this point, we have informally used the map in a refactoring course to help inexperienced developers understand and practice the overall refactoring philosophy, with very positive feedback.

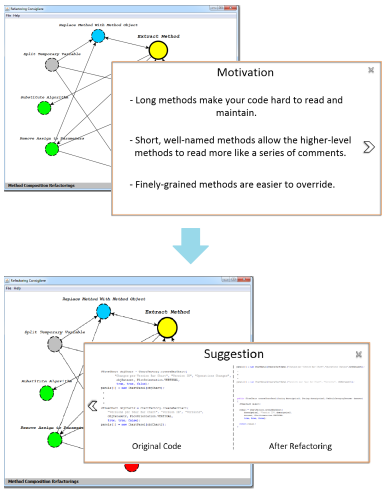

Figure 4: Overview of the refactoring trip advisor: The yellow node indicates the target refactoring, red nodes pinpoint refactorings that could be applied before the target refactoring, green nodes suggest refactorings to apply after the target one.

Our next steps focus on the development of the refactoring trip advisor. Figure 4, gives a sketch of the trip advisor's *modus operandi*. Starting from the refactoring the developer is considering, the trip advisor shall employ the map to detect possible alternative refactorings, and refactorings that could be done before and/or after, within the working code. Then, it will provide the developer with corresponding suggestions, along with a explicative discussion for the reasons for its advice and information about how to realize the refactorings and how the result would look like.

To this end, our first concern is usability; we are looking for scalable interactive, user-friendly map representations. Our second main concern is extensibility; we are after a flexible plug and play architecture that would allow (a) registering user-specific refactoring templates that combine elementary, Fowler-style refactorings (as related work suggests), and, (b) reusing existing mechanisms that detect refactoring opportunities. However, as already mentioned, the existing mechanisms cover only 27% of Fowler's refactorings, and it is also not sure that these mechanisms are suitable for an on the fly recommendation system for refactoring. Thus, we reach to the third important issue that completes our vision, the realization of fast, developer intuitive, heuristic-based mechanisms for the efficient detection of refactoring opportunities.

4. ACKNOWLEDGMENTS

This work received funding from the CHOReOS EU FP7 project no 257178.

5. REFERENCES

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

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

[3] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, and J. Sander. Decomposing Object-Oriented Class Modules Using an Agglomerative Clustering Technique. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 93–101, 2009.

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

[5] M. Harman and L. Tratt. Pareto Optimal Search Based Refactoring at the Design Level. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 1106–1113, 2007.

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

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

[8] E. Murphy-Hill, C. Parnin, and A. Black. How We Refactor, and How We Know It. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 287–297, 2009.

[9] F. Tip, R. M. Fuhrer, A. Kieżun, M. D. Ernst, I. Balaban, and B. de Sutter. Refactoring Using Type Constraints. *ACM Transactions on Programming Languages and Systems*, 33:1–47, 2011.