International Journal of Current Engineering and Technology
ISSN 2277 - 4106

Available at http://inpressco.com/category/ijcet

Research Article

# An Uncovering of Bad Smell in Software Refactoring

Ganesh B. Regulwar[a*] and R. M. Tugnayat [a]

[a]SSPACE, Wardha, Maharashtra, India

Accepted 10 October 2013, Available online 19 October 2013, **Vol.3, No.4 (October 2013)**

***Abstract***

*This paper discusses refactoring, which is one of the techniques to keep software maintainable. However, refactoring itself will not bring the full benefits, if we do not understand when refactoring needs to be applied. To make it easier for a software developer to decide whether certain software needs refactoring or not, Fowler & Beck (Fowler & Beck 2000) give a list of bad code smells. Bad smells are signs of potential problems in code. Detecting and resolving bad smells, however, remain time-consuming for software engineers despite proposals on bad smell detection and refactoring tools. Numerous bad smells have been recognized, yet the sequences in which the detection and resolution of different kinds of bad smells are performed are rarely discussed because software engineers do not know how to optimize sequences or determine the benefits of an optimal sequence. To this end, we propose a detection and resolution sequence for different kinds of bad smells to simplify their detection and resolution. We highlight the necessity of managing bad smell resolution sequences with a motivating example, and recommend a suitable sequence for commonly occurring bad smells. We evaluate this recommendation on two nontrivial open source applications, and the evaluation results suggest that a significant reduction in effort ranging from 17.64 to 20 percent can be achieved when bad smells are detected and resolved using the proposed sequence.*

***Keywords:*** *Scheme, bad smell, software refactoring, effort, detection, schedule.*

## 1. Introduction

### *1.1 Software Refactoring and Bad Smells*

Software refactoring is used to restructure the internal structure of object-oriented software to improve its quality, especially its maintainability, extensibility, and reusability (T. Mens et al, 2004), (http://books.google.co.in/books?id=1MsETFPD3I0C&dq=bad+smell+coding+in+software+refactoring+by+martin+fowler+%26+kent+beck&printsec=frontcover&source=in&hl=en&ei=nrVHTOJDIimvQOcxc3LAg&sa=X&oi=book), while preserving its external behavior. Software refactoring is widely used to delay the degradation effects of software aging and facilitate software maintenance. Because software is repeatedly modified according to evolving requirements, source code shifts from its original design structure. The source code becomes complex, difficult to read or debug, and even harder to extend. Software refactoring improves readability and extensibility by cleaning up bad smells in the source code.

A key issue in software refactoring is determining the kind of source code that requires refactoring. Experts have summarized typical situations that may require refactoring (Martin Fowler, 1999), (W.C. Wake et al, 2003). Fowler et al.call them Bad Smells (Martin Fowler, 1999), indicating that some part of the source code smells terrible. In other words, Bad Smells (e.g., Duplicated Code) are signs of potential problems in code that may require refactoring. These bad smells are usually linked to corresponding refactoring rules that can help dispel these bad smells.

### *1.2 Types Of Bad Smells*

The purpose of this section is to introduce those bad code smells, which are listed below:

**Long Method** is a method that is too long, so it is difficult to understand, change, or extend. Fowler and Beck (Fowler & Beck 2000) strongly believe in short methods. The longer a procedure is the more difficult it is to understand. Nearly all of the time all you have to do to shorten a method is *Extract Method*.

**Large Class** means that a class is trying to do too much. These classes have too many instance variables or methods, duplicated code cannot be far behind. A class with too much code is also a breeding ground for duplication. In both cases *Extract Class* and *Extract Subclass* will work.

**Primitive Obsession** smell represents a case where primitives are used instead of small classes. For example,

*Corresponding author **Ganesh B. Regulwar** is working as HOD CSE Dept. and **R. M. Tugnayat** is working as Principal

to represent money, programmers use primitives rather than creating a separate class that could encapsulate the necessary functionality like currency conversion.

**Long Parameter List** is a parameter list that is too long and thus difficult to understand. With objects you don't need to pass in everything the method needs, instead you pass in enough so the method can get to everything it needs. This is goodness, because long parameter lists are hard to understand, because they inconsistent and difficult to use because you are forever changing them as you need more data. Use *Replace Parameter with Method* when you can get the data in one parameter by making a request of an object you already know about.

**Lazy Class:** Each class you create costs money and time to maintain and understand. Lazy class is a class that is not doing enough and should therefore be removed. A class that isn't doing enough to pay for itself should be eliminated. If you have subclasses that are not doing enough try to use *Collapse*
*Hierarchy* and nearly useless components should be subjected to *Inline Class*.

**Duplicate code:** According to Fowler and Beck (Fowler & Beck 2000), redundant code is the worst smell. We should remove duplicate code whenever we see it, because it means we have to do everything more than once. If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them.

The simplest duplicated code problem is when you have the same expression in two methods of the same class. Perform *Extract Method* and invoke the code from both places. Another common duplication problem is having the same expression in two sibling subclasses. Perform *Extract Method* in both classes then *Pull Up Field.* If you have duplicated code in two unrelated classes, consider using *Extract Class* in one class and then use the new component in the other.

**Useless Field:** It is a synonym of Dead Field, referring to fields defined but never used.

**Useless Method:** Once a domain class is found, designers may propose methods (operations) for it. Unfortunately, sometimes certain operations are irrelevant to the role the class plays in the specific system. These operations are useless in the system.

**Useless Class:** Useless classes are those defined but never used. Typically, these are results of inappropriate boundaries of systems.

**Feature Envy:** A method or a fragment of a method may be more interested in features of another class
than those of the enclosing class, which is called feature envy.

### *1.3 Tool Support*

Tool support is crucial for the success of bad smell detection and resolution (Martin Fowler), (F. Tip et al, 2003), (Yoshio Kataoka ) because of the following reasons.

First, uncovering bad smells in large systems necessitates the use of detection tools because manually uncovering these smells is tedious and time-consuming, especially those involving more than one file or package, e.g. duplicated code. The tools are expected to detect bad smells automatically or semiautomatically. Clone detection is an excellent example, and researchers have proposed detection algorithms (R. Koschke et al, 2008), (T. Kamiya et al, 2002), and developed tools (T. Kamiya et al, 2002, for clone detection in the last decades.

Second, software engineers need tools to automatically or semiautomatically carry out refactorings to clean bad smells. Manual refactoring is time-consuming and errorprone. For example, renaming a variable requires revising all references to that variable. Manually identifying all references is challenging—issues that detection tools based on program analysis seek to address. Most mainstream integrated development environments (IDE), such as Eclipse,1 Microsoft Visual Studio, and IntelliJ IDEA , support software refactoring. Professional refactoring tools have also been developed. Rich tool support, in turn, accelerates the popularization of software refactoring. Human intervention, however, remains indispensable to bad smell detection and resolution because of the following reasons. First, most bad smells automatically detected should be rechecked manually because 100 percent precision cannot be guaranteed by detection tools.

Second, it is up to software engineers to determine how to restructure bad smells in terms of refactoring rules that should be applied, and arguments of the rule.

Finally, not all refactorings are supported by refactoring tools. For example, Eclipse (version 3.6, created on June 8 2010) supports only 23 refactoring rules but the number of proposed refactoring rules has increased to 93.

As a result, detecting and resolving bad smells remain time-consuming , even with tool support. Consequently, research aimed at simplifying detection and resolution is still urgently needed.

### *1.4 Detection and Resolution Sequence of Bad Smells*

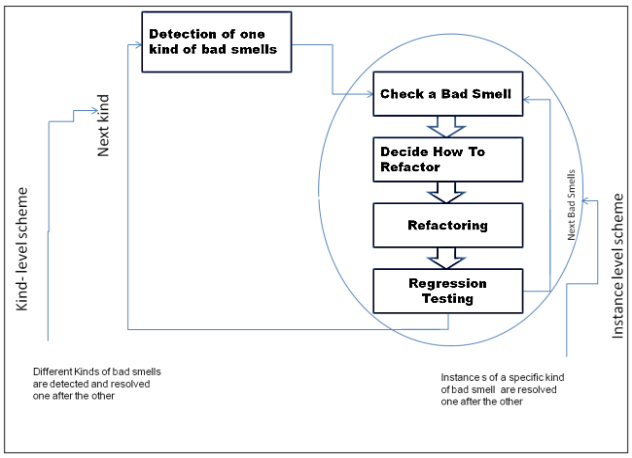


**Fig.1**. Detection and Resolution of Bad Smell

Software refactoring can be performed in two different

ways. The first is XP-style small-step refactoring confined to a few local files. The other is a batch model in which a large system is thoroughly refactored in one attempt.

In a batch model, different kinds of bad smells are typically detected and resolved individually. Suppose a software engineer, familiar with a list of bad smells and associated refactoring rules, refactors a large system. He is equipped with bad smell detection tools and automatic or semiautomatic refactoring tools for cleaning up bad smells. He first chooses a detection tool to identify a specific type of bad smell (a detection tool usually uncovers only one kind of bad smell, e.g., clone detection tools are insensitive to bad smells other than clones). The detection tool proposes initial results that require manual confirmation. Once the detected bad smell is confirmed, the software engineer decides how to refactor it. Selected refactoring rules are manually or semi automatically applied to the bad smells with the help of refactoring tools. Then, the software engineer moves on to the next kind of bad smells, and repeats the process until all kinds of bad smells have been detected and resolved. As a result, different kinds of bad smells are detected and resolved one after the other (Fig. 1), regardless of whether the sequence is arranged consciously or unconsciously.

The scheme for bad smells is two-tiered (Fig. 1): kind-level and instance-level. The kind-level scheme arranges the detection and resolution sequences of different kinds of bad smells. For example, should Large Class smells be detected and resolved before the Long Method type? Instance-level scheme arranges resolution sequences for instances of a specific kind of bad smell (e.g., Large Class). For example, numerous clones may be detected in a large application, and an instance-level scheme should manage the sequence in which these clones are resolved.

In this paper, we focus on the kind-level scheme.
Resolution of one kind of bad smells may influence (simplify or complicate) the detection and resolution of other bad smells. For example, Duplicated Code may cause Long method. Consequently, if Duplicated Code is resolved first, long method caused by it may disappear as well. However, little is known about the impact of resolving one kind of bad smell on the detection and resolution of other (remaining) bad smells.

Different resolution sequences of the same set of bad smells may require different efforts because resolving one kind of bad smell may simplify or complicate the detection and resolution of others. Therefore, it is possible to simplify bad smell detection and resolution by arranging appropriate detection and resolution sequences. The sequence in which Duplicated Code and Long method smells are resolved is an excellent example. Resolving Duplicated Code before Long Method is easier than the reverse because Long Method may disappear as a result of resolving Duplicated Code. Likewise, it is possible to maximize the effect of software refactoring by resolving bad smells in an optimal sequence. The importance of resolution sequences is illustrated in detail in Section 2.

To the best of our knowledge, thus far no published research focuses on the detection and resolution sequences of different kinds of bad smells (kind-level scheme in Fig. 1). Bouktif et al. attempted to schedule bad smell resolution. However, their study focused on the resolution sequence of different instances of the same kind of bad smell (Clone).Their approach is an instance-level scheme depicted in Fig. 1.

This paper offers the following contributions: We analyze the relationships among different kinds of bad smells and their influence on detection and resolution sequences. We identify the need to arrange detection and resolution sequences of different kinds of bad smells using a motivating example. We also recommend a resolution sequence for commonly occurring bad smells, and validate it on two nontrivial applications. The experimental results suggest that a significant reduction in refactoring effort (ranging from 17.64 to 20 percent) can be achieved when bad smells are resolved using the recommended sequence.

## 2. Resolution Sequences of Bad Smells

With the analysis results in Section 3, this section tries to recommend a resolution sequence for the evaluated bad smells. The algorithm is based on the following assumption: All bad smells of the same kind are scheduled as a single unit. In other words, all bad smells of the same kind would be detected and resolved before the next kind of bad smell is detected (as depicted in Fig.1). This assumption is based on the process of bad smell detection and resolution discussed in Section 1.3. As analyzed in that section, the assumption holds if refactorings are carried out in batch model.

First, we represent the analysis results of Section 3 with a directed graph (Fig. 2). Each vertex of the directed graph represents a category of bad smells whose name is presented in the label of the vertex. Each directed edge represents a preferred resolution sequence of the two kinds of bad smells represented by adjacent vertices of the edge, i.e., edge (v1, v2) indicates that v1 had better be resolved before v2.

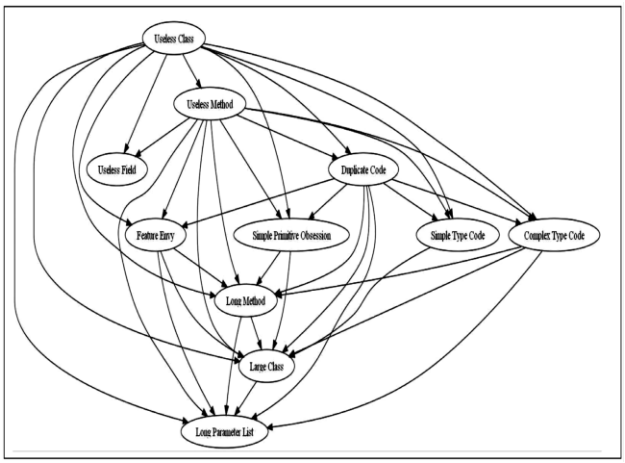


**Fig. 2** Original pair wise resolution sequences of bad smells.

Although the resulting graph is exhaustive, it is overly complicated to be presented to software engineers as

recommended guidance. Consequently, we simplify the graph by removing redundancies in the next step. We apply an algorithm to the directed graph to obtain a clear resolution sequence of different kinds of bad smells. One of the simplest ways to achieve the final resolution sequence is through topological sorting. However, when two or more vertices are available (whose in-degree is zero) in each iteration, topological sorting algorithms would randomly pick one of them for the next execution step. But we hope to leave the choice to software engineers who may take other factors into consideration, such as the severity of bad smells. Consequently, we propose a new algorithm for the final resolution sequences: removing redundant edges from Fig. 2.

For convenience, we define the following symbols:

- p (v1, v2): A path from vertex v1 to vertex v2 containing more than one edge.
- e (v1, v3): A direct edge from vertex v1 to vertex v3.

An edge e(v1; v3) is redundant if and only if there is another path p(v1,v3) in parallel to e(v1,v3). The graph in Fig. 3 is a good example. Removing edge e(v1,v3) would not change the topological order of the vertices, but removing any other edge from Fig. 3 would result in different topological order.

The algorithm removing redundant edges is presented in Fig. 4. If the algorithm is applied to the graph in Fig. 3, the algorithm would remove e(v1,v3) because the path p(v1, v2, v3) is parallel to e(v1,v3). Other edges would be retained because no path is longer than 1 in parallel to e(v1, v2), e(v2,v3), or e(v2,v4). The output of the algorithm on Fig. 4 is presented in Fig 5.

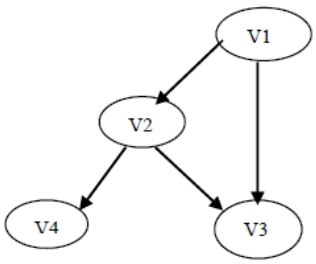


**Fig. 3**. Redundant edge

```
/*  Input: Directed graph with redundancies
        Output: Directed graph without redundancies */
For each vertex v in the graph
   {
     For each edge e(v,d) of vertex v
       {
If there is another path  p(v,d) besides e(v,d)
   {
//the length of p(v,d) must be longer than
//that of e(v,d) because there are no
//parallel edges in the graph
Remove e(v,d)
//removal of e(v,d )would not change
//the topological orders
   }
         }
     }
```

**Fig.4.**Algorithm for removing redundancies

The complexity of the algorithm is $O(|e|^2)$: O(|e|)for the two loops on lines 3 and 5, another O(|e|) for checking parallel paths on line 7. The complexity is not a huge problem. First, the graph is small and the algorithm can complete the computation quickly. On the nine kinds of bad smells analyzed in this paper (yielding a small graph with 38 edges), the algorithm completes the computation within 1 second (Window XP, Intel Core2 3.0 GHz, 2 GB memory). Second, the algorithm is run only once. Software engineers do not call the algorithm every time an application is refactored. Only when additional kinds of bad smells aside from those listed in Section 1.2 are introduced will the algorithm be called.

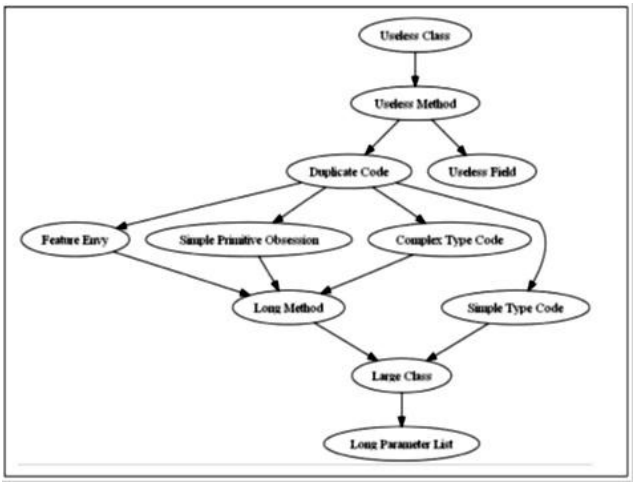


**Fig. 5.** Refined resolution sequences of bad smells.

The algorithm is deterministic, i.e., the same result is produced on the same input graph regardless of the order in which the edges and vertices are checked. The algorithm produces nothing but the longest paths between every pair of reachable vertices. For the graph in Fig. 3, the final result is a set of the longest paths between every pair of reachable vertices, i.e., <v1, v2>, <v1, v2, v3>, <v1, v2, v4>, <v2, v4>, <v2, v3>.

## 3. Related Works

### *3.1 Bad Smells*

To specify what kind of source code should be restructured, Fowler et al. (Martin Fowler, 1999) proposed the concept of bad smells. They proposed and described 22 bad smells in object-oriented systems. They also associated refactoring rules with these bad smells, suggesting how to resolve these bad smells.

Bad smells in specific domains have also been proposed. Srivisut and Muenchaisri defined some bad smells in aspect-oriented software, and proposed approaches to detect them. Van Deursen et al. described 11 Test Smells indicating problems in test code.

The impact of bad smells has also been analyzed. Lozano et al. assessed the impact of bad smells, i.e., the extent to which different bad smells influence software

maintainability. They argued that it is possible to analyze the impact of bad smells by analyzing historical information. With the impact in mind, it is possible to assess code quality by detecting and visualizing bad smells. Van Emden and Moonen implemented a code browser for detecting and visualizing code smells, and assessed the quality of code according to the visual representation.

Detecting bad smells is critical and time-consuming. Therefore, automating detection is essential. Tsantalis et al. proposed an approach to identifying and removing type-checking bad smells which is implemented in an prototype tool named JDeodorant. Fokaefs et al. [proposed an Eclipse plug-in to identify and resolve feature envy bad smells. Clones, one of the most common bad smells, have been investigated for a long time, and dozens of detection algorithms have been proposed (R. Koschke et al, 2008) to detect them. Moha et al. proposed a language for formalizing bad smells, and a framework for automatically generating detection algorithms for the formalized bad smells. Tourwe and Mens (T. Mens et al, 2004) formalized bad smells with logic metaprogramming and detected bad smells automatically. Munro related bad smells to software metrics to detect bad smells with automatically collected metrics. Van Rompaey et al. [proposed a set of metrics to detect test smells for unit testing. However, Mantyla et al. argued that detecting bad smells using metrics is difficult. In their experience, manual evaluation of bad smells does not correlate to source code metrics. Other detection algorithms can be found in the survey by Mens and Tourwe (T. Mens et al, 2004).

Trifu et al. [proposed a quality-guided approach to detect and remove design flaws. In their approach, a quality model is first proposed, and only those design flaws negatively impacting the quality model are detected. For each kind of design flaw, all alternative solutions are predesigned and their impacts on quality attributes are predefined. Once a design flaw is detected and confirmed, its alternative solutions are checked and the one that has greatest positive impact on software quality (according to the predefined quality model) is selected. The paper [20] differs from ours in that it focuses on which kind of bad smells should be detected and which refactorings should be applied, whereas this paper focuses on the sequence in which different kinds of bad smell should be detected and resolved.

### *3.2 Relationships among Bad Smells*

Relationships among bad smells have also been investigated. Wake (W.C. Wake et al, 2003) classified bad smells into two categories: bad smells within classes and bad smells between classes. Meszaros classified test smells into code smells, behavior smells, and project smells.

Mantyla et al. analyzed the correlations among bad smells by investing the frequency with which each pair of bad smells appears in the same module. They found that bad smells within the same category are more likely to appear together. The work aimed to simplify the comprehension of bad smells, instead of refactoring activities. The authors did not analyze the interinfluence of the resolutions of different bad smells, or the resolution sequences of bad smells.

Pietrzak and Walter investigated the intersmell relationships to facilitate the detection of bad smells. They argued that detected or rejected bad smells might imply the existence or absence of other bad smells. Their work aimed to simplify the detection of bad smells, whereas our work focuses on the detection and resolution sequences of different kinds of bad smells.

### *3.3 Scheme of Refactoring*

To make the scheme compatible with existing refactoring methods, this seminar tries to schedule the resolution of different kinds of bad smells. In this way, different kinds of bad smells are resolved one after the other. It cooperates well with existing refactoring tools: A detection tool is chosen to detect a specific kind of bad smell, the bad smells are resolved, and then the next detection tool is selected.

Wake (W.C. Wake et al, 2003) suggested resolving the most severe bad smells first, but he did not define the severity of bad smells. Furthermore, the strategy assumes that all reported bad smells will not be resolved. Software refactoring has been proved worthwhile, and we suggest carefully resolving all bad smells.

Bouktif et al. attempted to schedule clone refactoring activities. Because of resource limitation, not all clones can be removed. Consequently, they tried to select a subset of detected clones resolving which would lead to the greatest quality improvement while resource consumption is minimized. As illustrated in Fig. 1, the scheme proposed in this paper and the one proposed by Bouktif et al. are on different levels: The latter focuses on the resolution sequence of different occurrences of the same kind of bad smells (clone), whereas we focus on the detection and resolution sequence of different kinds of bad smells.

Mens et al. (T. Mens et al, 2004) realized that once a badly structured code is detected, many refactorings would be proposed. In this case, software engineers should decide which refactorings should be applied and in which sequence. To facilitate the decision, Mens et al. (T. Mens et al, 2004) proposed a critical pair-based approach to analyze refactoring dependencies. Their work [34] differs from ours in that they focused on XP style small step refactoring, i.e., a small code unit is checked manually by its author and all bad smells within the unit would be detected all at once. In this case, all bad smells can be resolved as a whole, whereas we aim at the situation in which a large system is thoroughly refactored in one attempt. In this case, tool support is indispensable in bad smell detection. Thus, different kinds of bad smells have to be detected and resolved one after the other.

In previous work, we have briefly discussed the resolution sequences of bad smells, but no evaluation or discussion was presented. This paper is an expanded version of that. In this paper, an evaluation on two

nontrivial applications is presented. Furthermore, the need for resolution sequences is illustrated in detail with a motivating example.Research in other aspects of software refactoring can be found in the survey by Mens and Touwe (T. Mens et al, 2004).

## 4. Conclusion

In this paper, we first motivated the necessity of arranging resolution sequences of bad smells with an example. Then, we illustrated how to arrange such a resolution sequence for commonly occurring bad smells. We also carried out evaluations on two nontrivial applications to validate the research. The results suggest a significant reduction in refactoring effort ranging from 17.64 to 20 percent can be achieved when bad smells are resolved using the recommended resolution sequence.

The contributions of this paper are as follows: First, it discovers and illustrates the importance of resolution sequences of bad smells. Second, it proposes a resolution sequence for commonly occurring bad smells. Finally, it validates the effect of resolution sequences of bad smells on two nontrivial applications.

## References

T. Mens and T. Touwe (Feb 2004), A Survey of Software Refactoring, *IEEE Trans. Software Eng.*, vol. 30, no. 2, pp. 126-139, Feb. 2004.

http://books.google.co.in/books?id=1MsETFPD3I0C&dq=bad+smell+coding+in+software+refactoring+by+martin+fowler+%26+kent+beck&printsec=frontcover&source=in&hl=en&ei=nrVHTOJDIimvQOcxc3LAg&sa=X&oi=book

Addison Wesley, 1999, Source: Martin Fowler, Refactoring - Improving the Design of Existing Code..

W.C. Wake (Aug 2003), Refactoring Workbook. *Addison Wesley* Refactoring : Improving the Design of existing Code :- Martin Fowler, Object Technology international, Inc , addison villey , *Pearson education*.

F. Tip, A. Kiezun, and D. Baeumer (Oct 2003), Refactoring for Generalization Using Type Constraints, *Proc. 18th Ann. Conf. Object- Oriented Programming Systems, Languages, and Applications*, pp. 13-26.

Yoshio Kataoka ,Tetsuji Fukaya , A quantitative evaluation of maintainability enhancement by Refactoring . *Proceedings of the international conference on software Maintenance(ICSM'02)IEEE*

R. Koschke (Sept 2008), Identifying and Removing Software Clones, Software Evolution, T. Mens and S. Demeyer, eds., pp. 15-36.

T. Kamiya, S. Kusumoto, and K. Inoue (July 2002), CCFinder: A Multi- Linguistic Token Based Code Clone Detection System for Large Scale Source Code, *IEEE Trans. Software Eng.*, vol. 28, no. 7,pp. 654-677.

Helsinki University of Technology, Department of Computer Science and Engineering, Software Business and Engineering Institute-Mika Mäntylä - Bad Smells in Software – a Taxonomy and an Empirical Study

E. Burd and J. Bailey (Oct 2002), Evaluating Clone Detection Tools for Use During Preventative Maintenance, *Proc. Second IEEE Int'l Workshop Source Code Analysis and Manipulation*, pp. 36-43.

Fowler, Martin: A list of refactoring tools for several languages, http:// www. refactoring. com/ tools.html

R. Wettel and R. Marinescu (2005), Archeology of Code Duplication: Recovering Duplication Chains from Small Duplication Fragments,Proc. Seventh Int'l Symp. Symbolic and Numeric Algorithms for Scientific Computing, p. 63.

Eclipse Foundation. Eclipse