

# Towards a Refactoring Guideline Using Code Clone Classification

Sandro Schulze    Martin Kuhlemann    Marko Rosenmüller

Department of Computer Science

University of Magdeburg

{sanschul,kuhlemann,rosenmueller}@iti.cs.uni-magdeburg.de

## Abstract

Evolution of software often decreases desired properties like readability and maintainability of the evolved code. The process of refactoring aims at increasing the same desired properties by restructuring the code. New paradigms like AOP allow aspect-oriented refactorings as counterparts of object-oriented refactoring with the same aim. However, it is not obvious to the user, when to use which paradigm for achieving certain goals. In this paper we present an approach of code clone classification, which advises the developer when to use a respective refactoring technique or concept.

**Categories and Subject Descriptors** D2.7 [Distribution, Maintainance, and Enhancement]: Restructuring, Reengineering

**General Terms** DESIGN, LANGUAGES

**Keywords** Refactoring, clone detection, object-oriented programming, aspect-oriented programming

## 1. Introduction

Object-oriented software systems are known to evolve during their whole lifetime. This evolution leads to several anomalies (*code smells*) decreasing desired system properties like readability or maintainability. *Refactorings* allow to re-increase these desired properties by changing the internal structure of software while preserving its external behaviour [7, 13].

Recent research on refactorings mainly tackles problems of when and how to apply *Object-Oriented Refactorings (OOR)* [7, 10]. However, with the emergence of *Aspect-Oriented Programming (AOP)* (amongst other new programming paradigms), new refactorings using AOP mechanisms arose, resulting into *Aspect-Oriented Refactoring (AOR)*. Unfortunately, if a system is suffering from code smells, it

is left to the user to select a proper paradigm and its corresponding refactoring to solve this anomaly [7]. Additionally, the decision is hindered by the fact that both *refactoring concepts*<sup>1</sup>, OOR and AOR, are applicable alternatively to certain code smells.

In this paper, we present our idea of using code clone classification to support the decision whether to use OOR or AOR for removing every single code smell/clone. With this approach we support the refactoring process by adding semantic information to code clones in order to derive appropriate refactoring techniques. Code clones [12], i.e., identical or similar code fragments replicated across the code base, are a special kind of code smells and beyond, they can be seen as *homogeneous crosscuts* [1], which plays to the strength of AOP.

## 2. Background

### 2.1 Code Smells/Code Clones

Code smells are code structures that suggest the possibility of refactorings [7]. Replicated code, a special kind of code smell, is considered to be a serious and widespread problem. Other researchers [2, 3, 6] showed that up to 15 % of the code suffers from code clones, which decreases desired properties like maintainability or readability. Thus, it is worthwhile to improve the refactoring process of clones. Although current OO programming languages provide different reuse-mechanisms [8, 14], code cloning is still common. Code clones occur when a software's design evolves improperly due to (1) hard time constraints for the developer evolving the system, (2) little knowledge of a developer about the system, or (3) crosscutting concerns [6, 4, 5].

### 2.2 Aspect-Oriented Programming

Aspects in AOP define code that should be inserted at different points of an OO base program. Two of the main AOP mechanisms used for that are introductions and pointcut-advice.

Copyright is held by the author/owner(s).

WRT'08, October 19, 2008, Nashville, Tennessee, USA.  
ACM 978-1-60558-339-6/08/10.

<sup>1</sup>We refer to an overall refactoring idea (encompassing the used paradigm and concrete refactorings amongst others) as refactoring concept and to a concrete refactoring (e.g., Extract Method) as refactoring (technique).

Introductions add methods and fields to classes. Pointcut mechanisms quantify *join points* over different code fragments of the base program that should be extended; advice is the code added to these code fragments. Pointcut-advice mechanisms thus can modularize code scattered across completely unrelated classes.

### 3. Problem Statement

The presence of two possible *refactoring concepts*, OOR and AOR, provokes the decision problem *what* concept to be used (besides the problem which concrete refactoring technique to apply) for code smells. Existing approaches discuss the refactoring process itself and how to embed refactoring in other software engineering processes (e.g., reverse engineering) or tools and frameworks (e.g., eclipse) and automating the mechanics of refactorings. New concepts, methods, and paradigms are challenges to existing and approved approaches and have to be compared and distinguished. Additionally, it is still an entirely open issue how to determine an appropriate refactoring (concept *and* technique) for a certain kind of code smell, e.g., duplicated code. For instance, in spite of existing techniques and tools for the detection of code clones, all end up with information about their presence, e.g., in form of visualizations or metrics. The next step, which is establishing a relation between detected code clones and suitable refactorings for their removal, is missing in every case (to the best of our knowledge).

### 4. An exemplary Refactoring Scenario

Now we give a simple motivating example for choosing the correct refactoring based on reasoning. In Figure 1 a method is listed, which we detected as replicated code within the JBoss Application Server<sup>2</sup>. More precisely, this method occurred in identical form (differing only in the name of variables) several times in the source code. With the aim of eliminating this code fragment and all its copies, the question raises, *how* to remove them. Answering this question leads exactly to the problem we tackle.

```
1 public String getEventType(int pIndex) {
2     int tmp = StateManagement.stateTypes.length;
3     if (pIndex >= 0 && pIndex < tmp)
4     {
5         return StateManagement.stateTypes[pIndex];
6     }
7     else
8     {
9         return null;
10    }
11 }
```

Figure 1. Exemplary code fragment

Suppose that the corresponding code clones are scattered across several classes, exhibiting the same superclass. Subsequently, applying the *Pull Up Method* refactoring [7] is the

<sup>2</sup><http://www.jboss.org/jbossas/>

most suitable solution. In the case that the several clones are slightly different, extracting the identical part and pulling it up is conceivable as well. However, if the code clones are not tightly bounded to each other, e.g., their classes have no common superclass or even are not apparent in the same component, AOR seems more convenient, where an aspect is known to crosscut the whole software. Potentially, the application of OOR may cause spread of code regarding one concern across packages and directories which lacks cohesion.

Alternatively, the code could be moved into an aspect, where aspects are known to modularize code cutting across the whole product [11].

However, specifying the appropriate refactoring by hand is a time-consuming process, prone to errors, e.g., a code clone could be missed. Thus, a guideline is needed which supports the developer in specifying and applying the suitable refactorings.

### 5. When to Use OOR and AOR

In the following we present a classification of code clones and propose a guideline which refactoring should be used in order to remove clones for each category and thus, improve modularity and maintainability of the system.

#### 5.1 Classification of Code Clones

Our classification aims at adding semantic information to code clones which can be used as a guideline of appropriate refactoring techniques for code clone removal. The code fragment and all its clones are encompassed under the term of a *clone class* [9]. The classification takes place along two dimensions, *Type* and *Location*.

*Type* specifies the kind of statement of the clone classes (i.e., *conditionals*, *loops*, *functions*, and *other*) and has been chosen for two reasons. First, the subcategories (e.g., *functions*) refer to code blocks which are prone to code replication because they often comprise a piece of functionality or an identical semantic idea, easy to reuse by copying (and adaptation, if needed). Second, a wide range of refactoring techniques exists for the respective subcategories, which enables a behaviour-preserving elimination of the code clones.

The *Location* attribute reflects the distribution of corresponding code clones with respect to the file system (and thus, to the project structure as well). This attribute (and its categorization) has been chosen with respect to the structure commonly used to arrange source code. When building software, the correlations between classes, modules, or components are mostly reflected by their physical and logical arrangement. Thus, files implementing highly related functionality, are often placed in one directory or several sub-directories of a common directory. In contrast, files implementing non-related functionality are scattered throughout the whole system or software project respectively.

The information for classifying clone classes based on the location of their clones can be obtained by a metric. For the sake of brevity, we just mention the main characteristics of a metric, called DIST, we set up. The definition of this metric is as follows:

$$DIST = scale * (w_{vert} * vGap_{cc} + w_{hor} * hGap_{cc}) \quad (1)$$

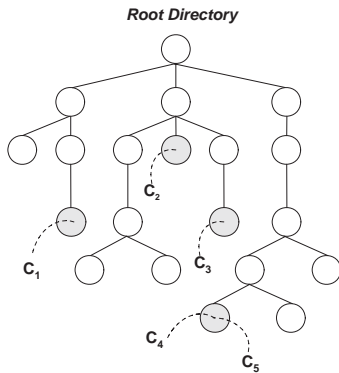
The two parameters  $vGap_{cc}$  and  $hGap_{cc}$  determine the distance of the code clones within a clone class in vertical and horizontal direction respectively. They are calculated using the  $vGap$  and  $hGap$  values for each *clone pair* of the clone class and are defined as follows for a clone pair  $(C_A, C_B)$ :

$$vGap(C_A, C_B) = |depth(C_A) - depth(C_B)| \quad (2)$$

$$hGap(C_A, C_B) = \max(vGap(C_A, CNode), vGap(C_B, CNode)) \quad (3)$$

$depth(\dots)$  is the difference between the code clone and the root node while  $CNode$  determines the first common node of two code clones. Amongst all computed values for  $vGap$  and  $hGap$ , the particular maximum has to be determined leading to the parameters  $vGap_{cc}$  and  $hGap_{cc}$  respectively.

For specifying these parameters, an *Abstract Directory Tree (ADT)* is used, representing the internal file (or project) structure of the system. A sample tree is depicted in Figure 2, containing five code clones  $C_1, \dots, C_5$ . The root of the tree represents the root directory of the considered software project. Underneath the root, several levels can follow, consisting of nodes that represent directories as well as source files. The leaf nodes only represent source files. In the context of the directory tree, a source file is nothing else than a container of zero or more code clones. The parameter values for the depicted clone class are  $vGap_{cc} = 3$  and  $hGap_{cc} = 5$ .



**Figure 2.** Tree structure used for parameters  $hGap$  and  $vGap$ .

Furthermore, both parameters can be weighted, which is indicated by  $w_{vert}$  and  $w_{gap}$  in Equation (1). Since the parameters are balanced weighted by default, adjusting them

can be useful in the case of a heavily unbalanced ADT, e.g., a very broad and shallow ADT or a slim and deep one. Additionally, the DIST metric is normalized, indicated by the scaling factor  $scale = \frac{1}{\max(depth(C_1), \dots, depth(C_n))}$ . Hence, we achieve a finite range for our metric,  $0 \leq DIST \leq 1$ . This range allows a straightforward classification and guideline for refactorings.

## 5.2 Guiding the Refactoring Process

With the help of the introduced classification, we can now provide a guideline for refactorings to be used. We mainly use the metric for specifying the appropriate refactoring paradigm (or concept) while the *Type* is used to specify the concrete refactoring technique. As a first step, we divide the range of our metric into three partitions, based on our experience with the JBoss Application Server. The first partition,  $0 \leq DIST < 0.3$  contains clone classes whose members (i.e., the code clones) are highly related to each other. The partition  $0.6 \leq DIST \leq 1$  indicates that the code clones are scattered across the source code. Finally, a middle partition  $0.3 \leq DIST < 0.6$  is the most critical one, regarding the refactoring proposal, because a universal conclusion about these clone classes cannot be made (at first glance).

Now, with the categorized metric results, we can provide a guideline for code clone removal using refactorings. An overview, encompassing suitable refactorings with respect to the classification is given in Table 1. The table shows, that our guideline is restricted to seven refactorings. We concentrate on these seven refactorings because the problems they solve occur most frequently compared to others (regarding code replication). According to the table, all clone classes of the first partition should mostly be removed by using techniques from the OOR concept, differing only in the concrete technique, which depends on the *Type* attribute. This is caused by the low DIST value, which indicates a tight relation between the corresponding code clones in form of a common superclass or similar.

For the second partition, giving a guideline is not as easy as for the first one. Here our approach exhibits still some weaknesses in the sense, that certain conditions have to be checked manually for specifying the most suitable refactoring. The reason is, that we cannot point out exactly, how tight the relation between corresponding code clones is. In general, our experience has been shown that clone classes close to the lower limit of this partition tend to be tightly related and thus, one of the OO refactorings (depending on the *Type*) is appropriate for their removal. However, all other clone classes of this partition are now subject to AOR and one of its concrete techniques.

Regarding the last partition, the refactorings proposed by our guideline have completely shifted towards the aspect-oriented paradigm. The code clones of this partition are loosely related (regarding their occurrences within the file system) and thus, crosscut the whole source code. Subse-

	Conditional/Loop					Function				
	0.0	< 0.1	< 0.3	< 0.6	< 1.0	0.0	< 0.1	< 0.3	< 0.6	< 1.0
Extr. Meth.	++	++	+	-	--	+	-	-	--	--
Pull Up/Move Meth.	--	++	+	-	--	--	++	+	-	--
Form Templ. Meth.	--	--	--	--	--	++	++	+	+	+
Extr. Feature into Aspect	--	--	+	++	++	--	--	+	++	++
Extr. Fragment into Advice	--	--	+	+	+	--	--	+	+	+
Move Method from Class to Inter-Type	--	--	+	+	+	--	--	+	+	+

**Table 1.** Refactoring techniques and their relation to the classification.

++ *approved w/o constraints*, + *approved with constraints*, - *in exceptional cases*, -- *not suitable/applicable*

quently, extracting them into an aspect via AOR is useful to increase modularity. The concrete refactoring depends not only on the *Type* attribute but also on the fact, whether a certain aspect already exists, e.g., implements a functionality similar to that of the considered clone class.

## 6. Conclusions

In this paper, we presented our idea of a guideline of refactorings for code clone removal regarding OOR and AOR. Therefore, we presented a classification which adds semantic informations to code clones. Using the classification results, especially of a metric we built up, we proposed appropriate refactorings for each category of code clones. This guideline can be used for their removal. However, since the guideline is already useful for specifying refactorings for code clone removal there is still some work to be done. First, for the final refactoring process, it is still necessary to use a separate refactoring tool or an IDE which supports (semi-)automated refactoring (e.g., Eclipse as we did). Currently we are working on tool support, integrating classification and semi-automated refactoring. Furthermore, we intend using an *Abstract Syntax Tree (AST)* instead of the ADT so that we can specify the relation between corresponding clones more precisely, e.g., by deploying the inheritance hierarchy, comparison of ingoing/outgoing methods or access of fields.

## Acknowledgments

One of the authors is funded by the European Union under contract number C(2007)5254.

## References

- [1] S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. In *IEEE Transactions on Software Engineering (TSE)*, 2008.
- [2] Brenda S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, 1995.
- [3] I.D. Baxter et al. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, 1998.
- [4] M. Bruntink, A.v. Deursen, R.v. Engelen, and T. Tourwé. On the Use of Clone Detection for Identifying Crosscutting Concern Code. In *IEEE Transactions on Software Engineering (TSE)*, 2005.
- [5] E. Duala-Ekoko and M.P. Robillard. Tracking Code Clones in Evolving Software. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2007.
- [6] S. Ducasse, M. Rieger, and S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, 1999.
- [7] Martin Fowler et al. *Refactoring - Improving the Design of Existing Code*. Addison Wesley, 2000.
- [8] R. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2), 1988.
- [9] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. In *IEEE Transactions on Software Engineering (TSE)*, 2002.
- [10] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2004.
- [11] M.P. Monteiro and J.M. Fernandes. Towards a Catalog of Aspect-Oriented Refactorings. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, 2005.
- [12] C.K. Roy and J. Cordy. A Survey on Software Clone Detection Research. Technical Report 2007-451, School of Computing, Queen's University at Kingston, 2007.
- [13] L. Tokuda and D. Batory. Evolving Object-Oriented Designs with Refactorings. In *Proc. of the Int'l Conf. on Automated Software Engineering*, 2001.
- [14] Peter Wegner. Concepts and Paradigms of Object-Oriented Programming. *ACM Special Interest's Group of Programming Languages (SIGPLAN)*, 1990.