Tool Support for Refactoring Duplicated OO Code

Stéphane Ducasse and Matthias Rieger and Georges Golomingi
Software Composition Group, Institut für Informatik (IAM)
Universität Bern, Neubrückstrasse 10, CH-3012 Berne, Switzerland
{ducasse, rieger, goloming}@iam.unibe.ch
http://www.iam.unibe.ch/∼scg/

Position Paper, April 13 1999

Abstract

Code duplication is an important problem in application maintenance. Tools exist that support code duplication detection. However, few of them propose a solution for the problem, i.e. refactorings. We propose an approach that uses the information given by code duplication detection to guide the refactorings of OO applications.

1 Code Duplication and Refactorings: Two Poles of a Magnet

Code duplication is a well-known problem in software maintenance, bloating up source code and complicating the extermination of errors. Tool support exists that helps to identify the duplicated code [Joh93, Bak92, BYM$^+$98, DRD99]. Refactorings are a way to transform an application in a behaviour preserving manner [Opd92]. Refactoring tools [RBJ97] support the elementary operations to transform code. Duplication is one important aspect of code where refactorings can be used to improve the source code.

Very few tools, however, support corrections of systems plagued by duplicated code. In [Bak97, BYM$^+$98], macro extraction from source code (C and C++) were proposed, but these approaches did not tackle the intricate problems of refactorings in an object-oriented context.

In this paper, we propose to use a duplication detection tool to guide the refactoring process. We want to stress the fact that our proposition is not to fully automate the process but to help the reengineer by doing a first analysis of the situation and providing him with a solution if possible. This paper presents first an overview over the

[0]Appeared in "Object-Oriented Technology (ECOOP '99 Workshop Reader)", LNCS 1743, Springer Verlag, 1999

ingredients that we need for our idea and then gives a short overview how duplication in object-oriented code can be categorised and tackled.

2 Combining Duplication Information and a Static Object-Oriented Model

The first ingredient for our approach is obviously a tool to detect duplication in source code. Secondly, the approach is based on the ability to represent a program in terms of object-oriented entities, like for example classes, methods and attributes (see section 2.2). To represent the program in such a model is crucial since the possible refactorings that can be carried out on duplication found in object-oriented code depend on the structural relationships of the program entities the duplication was found in.

In this section, we first detail the two ingredients, then propose a step by step recipe which combines them for the goal of guiding refactorings.

2.1 Identifying Duplication

DUPLOC is a tool for detecting duplicated code [RD98, DRD99]. DUPLOC has been developed in the context of the FAMOOS Esprit project and features:

- a code reader that is easily adaptable to different programming languages.
- a line based comparison which uses basic string matching. The result of a match, a boolean true or false, is recorded in a two dimensional matrix.

What is interesting for refactorings are matches that span multiple source lines, e.g. parts of methods or even entire methods. To extract this information from the line based comparison matrix, pattern matching is used to condense the duplication information to sequence entities.

DUPLOC does not divide source code into programming language entities. Since it wants to remain language independent it cannot use parsing to separate classes or methods in the source code. Comparing two programs thus results in one big, unstructured comparison matrix. DUPLOC does however adhere to the ‘physical’ structure of source code, namely to files (called *source objects*). The comparison matrix is divided into comparison between two source objects.

DUPLOC provides two ways of reporting on the duplication found: a) it uses scatter plots to visualise the duplication on screen b) it produces textual reports indicating the locations of the sequences.

Duplication Information provided by DUPLOC

The report extract shown below presents matched sequences that were found in two comparisons. The structure of the report is the following: First, the participants of the comparison are identified. Then, a summary of all the sequences found is presented. Finally, all matched sequences are listed in detail. In the `Pattern`-string of a match, a vertical bar stands for two lines that matched, and a horizontal bar marks two lines

```
Comparison of
~/Cobeakti.cls
with
~/Cobegtyp.cls

Sequence length: 11 Number of Sequences: 1

A Match between code from file 'Cobeakti.cls' (from line 43 to line 63)
[Pattern: |.-.-.|.|.|.|.|.|.|.|]
and code from file 'Cobegtyp.cls' (from line 133 to line 153)
[Pattern: |.-.-.|.|.|.|.|.|.|.|].
(Stretch = 11 , RelevantLines = 9)
```

Figure 1: An example for duplication information produced by DUPLOC.

that did not match. A dot in the pattern string represents a source line that contained comments or white space and was removed from the input before the comparison. Such lines are not taken into account during the pattern matching and are printed so that reported source line numbers correspond to the length of the pattern.

2.2 A Meta Model Supporting Reengineering

The FAMIX meta-model, also developed in the FAMOOS project, is a language independent representation of object-oriented sources [DTS98].

The Core Model. The core model (see figure 2) specifies the entities and relations existing in an object-oriented program. The core model consists of the main OO entities, namely Class, Method, Attribute and InheritanceDefinition. For reengineering purposes we need the other two, the associations Invocation and Access. An Invocation represents a Method calling another Method and an Access represents a Method accessing an Attribute. The complete specification of the model, including abstractions for extending the model, can be found in [DTS98].[1]

For a specific system, instances of these entities are extracted directly from source code and stored in a database. Each entity keeps a source anchor to the location it was extracted from.

2.3 The Approach Step-by-Step

To refactor applications based on duplication information we combine the tools presented in the previous sections in the following process:

1. We represent the object-oriented application we want to analyse using a language-independent meta model. The entities of the meta model contain source anchors.

[1]The complete model is more general: an Invocation is about behavioral entities (such as methods and functions) calling other behavioral entities, and an Access is about a behavioural entity accessing a structural entity (such as instance variables and global or local variables).

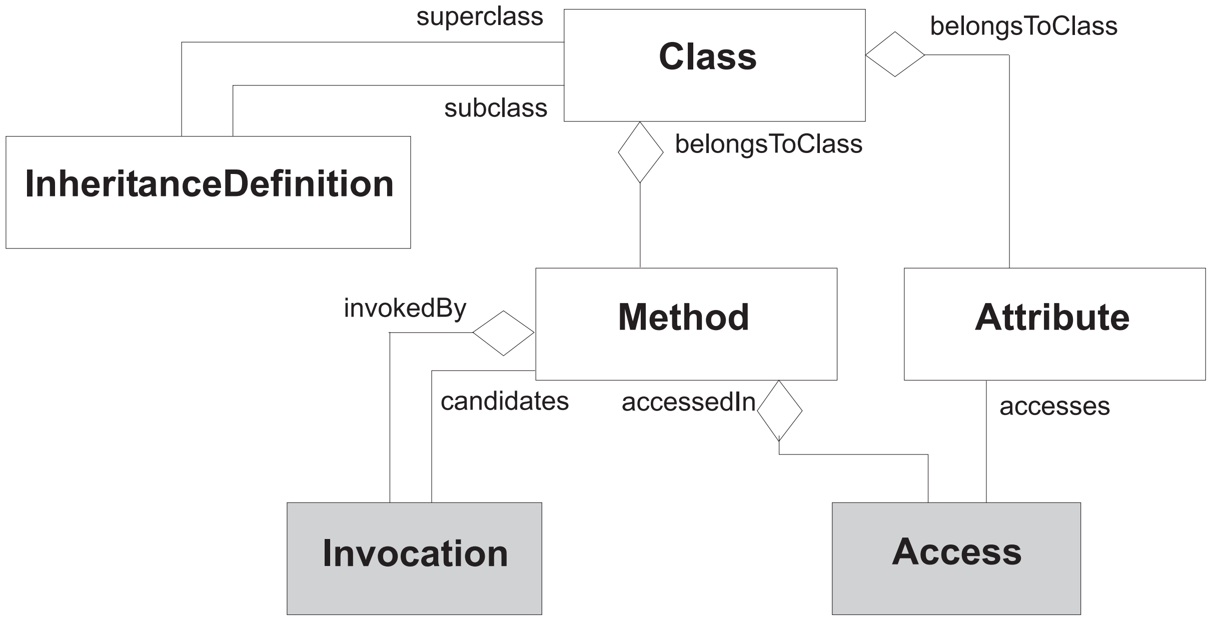

Figure 2: The Core Model

2. According to an object-oriented refactoring scenarios (e.g. refactorings inside a single class, between sibling classes, or between unrelated classes, see section 3) entities from the application model are selected and their source code is fetched forming *source objects* for DUPLOC.

3. The duplication information is generated by running the source objects through DUPLOC.

4. Analysis of the duplication in the context of the refactoring scenario selected in step 2: can the duplication be refactored? Note that this step requires the interaction with a human operator.

Note that—although it is not the point we want to prove with this paper—our approach is to a large extent language independent: the meta-model shields the language from us and DUPLOC does not depend on the language as well. Note also that to actually perform the refactorings in an automated fashion, we need a tool that is able to manipulate code in the form of abstract syntax trees. This is where language independence gets lost again. In our current setting we are staying in the SMALLTALK domain where we can delegate the execution of the refactorings to the Refactoring Browser.

3 Common Duplication Situations in OO Code

The topic of refactoring duplication in object-oriented source code can be divided in two orthogonal subtopics: 1) refactoring operations on duplicated code in general (see section 3.1), and 2) refactoring situations in an object-oriented context (see section 3.2).

3.1 Refactoring Operations to Counter Code Duplication

In our analysis of duplication we see two cases where refactoring operations can be applied:

1. A complete function is duplicated exactly, for example functions F, F', . . . , $F'^{\cdots\prime}$. In this case the solution is to change the calls to the functions F', . . . , $F'^{\cdots\prime}$ into calls to the function F, and then remove all duplicates but F.

2. A piece of code is found as part of a number of functions. The solution is to extract the piece and create a new function N from it. The duplicated code is then replaced everywhere with calls to N.

Having identified a duplication situation it is however not always possible to apply the refactorings in a straight forward manner. Additional analysis is needed to determine the context in which the duplicated code is found. The context consists in local variables and parameters that are accessed by the duplicated code. This analysis is complicated even more in object-oriented code.

When confronted with a piece of code from a method of a class, the following questions must be asked:

- Does the code contain references to the parameters of the method it is part of?
- Does the code contain references to temporary variables of the method?
- Does the code contain references to instance variables of the class?
- Which access restrictions (*public*, *protected*, or *private*) constrain the method?
- Is the method overridden in a subclass?
- Does the method override a method in the superclass?

It is out of the scope of this short paper to go into detailed analyses of these cases. We only try to hint at the kind of support that can be expected from tools. Tools should support the listing of dependencies and the testing of preconditions before a refactoring operation is applied. In simple cases, the tool can propose refactorings. In other cases, the selection of the operation must be left to the human operator.

3.2 Refactoring Duplication in an Object-Oriented Context

Refactoring in an object-oriented context is constrained by the relationships between the software entities. Three possible relationships and the refactorings that can be applied in each of the cases are explained in this section.

Duplication inside of a Single Class

The most straight forward case is when a part of the code is duplicated inside a single class. To execute the refactorings, no special care has to be taken about class borders, the whole class can be seen as a procedural program.

Duplication between Sibling Classes

By *sibling classes* we refer to all classes that have the same superclass and are at the same hierarchy level. For the sake of a first simple analysis, we do not include classes at different hierarchy levels. To further simplify the analysis, we only consider situations where *all* the subclasses of a common superclass contain the duplication. The solution in this case is to move the the one method that has been left over after the refactoring has been executed to the common superclass.

When only a subset of the classes contains the duplication, a possible solution is to insert an intermediary class into the hierarchy which contains the factored out code.

Duplication between Unrelated Classes

Proposing a solution for this situation is the most difficult. If the same functionality is found in different classes, this might indicate a new responsibility which could be factored out into its own class.

4 Conclusion

Refactoring duplicated code can be supported by tools. What is essential for refactoring object-oriented code is a meta-model with which to represent the application. Duplication information can only be interpreted within such a model of the application. The automation is possible to a full extent only for some very simple cases of exact cloning. Support for more complicated cases is constrained to dependency analysis and precondition checking. The final authority in these cases will always be the programmer.

References

[Bak92] Brenda S. Baker. A Program for Identifying Duplicated Code. *Computing Science and Statistics*, 24:49–57, 1992.

[Bak97] Brenda S. Baker. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance. *SIAM Journal of Computing*, October 1997.

[BYM$^+$98] Ira Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant' Anna, and Lorraine Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of ICSM*. IEEE, 1998.

[DRD99] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A Language Independent Approach for Detecting Duplicated Code, 1999. to appear in ICSM'99.

[DTS98] Serge Demeyer, Sander Tichelaar, and Patrick Steyaert. Definition of a common exchange model. technical report, University of Berne, July 1998.

[Joh93] J. Howard Johnson. Identifying Redundancy in Source Code using Fingerprints. In *Proceedings of CASCON'93*, pages 171–183, 1993.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.

[RBJ97] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for smalltalk. *Journal of Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[RD98] Matthias Rieger and Stéphane Ducasse. Visual detection of duplicated code. In Stéphane Ducasse and Joachim Weisbrod, editors, *Proceedings ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*, number 6/7/98 in FZI Report. Forschungszentrum Informatik Karlsruhe, 1998.