Beyond the Refactoring Browser: Advanced Tool Support for Software Refactoring

Tom Mens Tom Tourwé Francisca Muñoz
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
Email: { tom.mens | tom.tourwe | fmunozbr }@vub.ac.be

Abstract

Current refactoring tools only provide support for performing selected refactorings. We show how tool support can be provided for the preparatory phases of refactoring as well, by determining when a software application should be refactored and which refactoring(s) in particular should be applied. We implemented a tool to detect bad smells and to propose adequate refactorings based on these smells, and validated this tool by carrying out experiments in three concrete case studies: the Soul application, the Smalltalk Collection hierarchy, and the HotDraw application framework. We also show how our tool complements the Smalltalk Refactoring Browser.

1 Introduction

Refactoring is the process of improving the structure of an object-oriented application without changing its overall behaviour [9]. Although the definition of refactoring has been around for several years, its importance in object-oriented development and reengineering has only recently been acknowledged. Most major integrated development environments for object-oriented programming languages incorporate support for refactoring, and refactoring is more and more discussed in the context of reengineering legacy applications [5, 12].

We can identify three distinct steps in the refactoring process: (1) detect when an application should be refactored; (2) identify which refactorings should be applied and where; (3) perform the refactorings. The support offered by most current development environments is limited to step 3. Most tools present a list of refactorings to the developer, and upon selection of any refactoring from this list, automatically perform the corresponding changes. Although this relieves the developer from the difficult and error-prone process of performing these changes manually, it still requires him to apply the earlier identification steps by hand.

In order to find out which parts of the source code need to be refactored, we suggest to rely on the notion of *bad smells*. Originally coined by Kent Beck [6], the term refers to *structures in the code that suggest (sometimes scream for) the possibility of refactoring*. Once bad smells have been identified, refactorings need to be proposed to resolve these smells. Automated support is crucial here, due to the large number of refactorings that are available.

In this paper we propose an advanced refactoring tool that automates the three steps of the refactoring process. Step 1 identifies bad smells in the program code. Step 2 proposes refactorings to remove these bad smells. Step 3 uses the existing refactoring browser to apply the proposed refactorings, after manual inspection by the programmer.

As a proof of concept, we implemented our tool in the *VisualWorks Smalltalk* object-oriented programming environment, of which the *Refactoring Browser* [10] is an integral part. We show how our tool complements, and is integrated with this browser, and illustrate the added value of our tool by means of a number of experiments performed on three realistic case studies. We also discuss some scalability issues we encountered when carrying out our experiments.

2 Advanced Refactoring Tool Support

The tool we propose has a straightforward and easy to use interface. Rather than being fully automated, the tool is driven by the programmer, who selects a particular source code entity to be analysed for bad smells, and then chooses one of the proposed opportunities for refactoring. This then triggers the refactoring engine that applies the selected refactoring.

To provide this kind of support, we integrated our tool with the *Refactoring Browser* of *VisualWorks Smalltalk*, which already provides support for applying user-selected

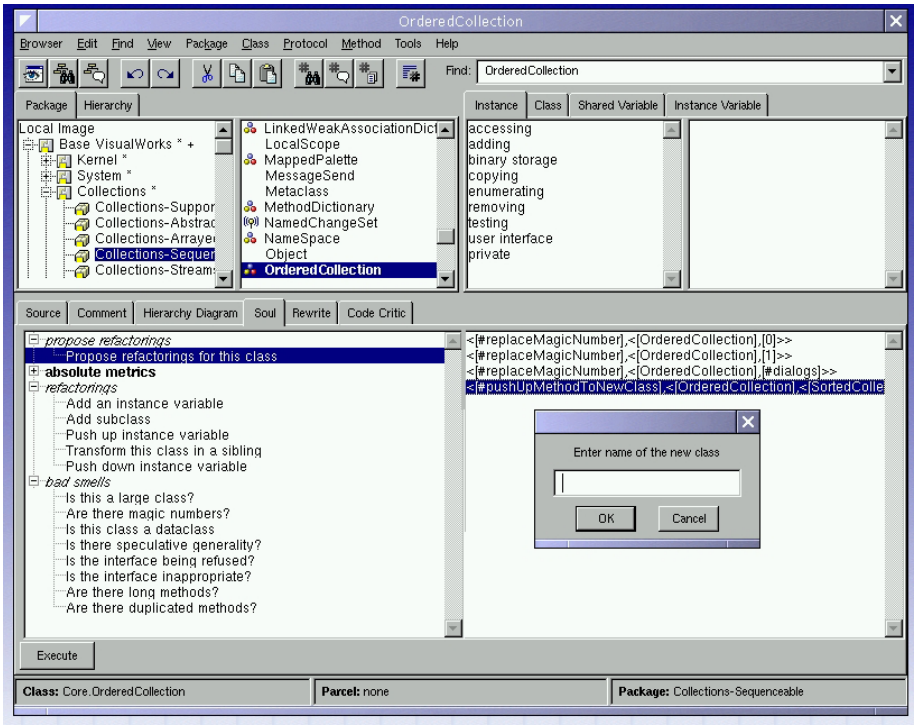

Figure 1. Tool support for detecting bad smells and proposing refactorings

refactorings. We augmented this browser with a *Soul* tab (see Figure 1), which exists next to the other tabs already available (such as the *Source*, *Comment* and *Code Critic* tabs). The *Soul* tab offers a list of logic queries that can directly be invoked by the user from within the *Smalltalk* browser. Upon selection of a class in the upper left pane, the developer can select a query and click on the *Execute* button. As a result, the logic query will be executed and the results will be shown in the lower right pane.

As can be seen in Figure 1, our tool offers several categories of queries, and the developer is free to use any combination of them in any order:

Bad smells. This category contains all the queries that we implemented for detecting a particular bad smell. The result of applying a bad smell query is a list of *Smalltalk* entities (e.g., classes or objects) that are qualified with their bad smell. As an example, *Is there speculative generality?*, one of the bad smells enumerated by Kent Beck and Martin Fowler in [6], detects a number of different problems, including the fact that a method has unused parameters.

Refactorings. This category provides an alternative way to access the refactoring engine of *Smalltalk*. Most of the queries in this category are wrappers that execute the corresponding refactoring in the *Refactoring Browser*. For example, *Add subclass* invokes the built-in refactoring that creates a new subclass of the current class, and allows the user to select which of the current subclasses should become children of the new subclass. If desired, more specific preconditions may be specified for each refactoring, or user-defined refactorings may be added.

Propose refactorings. This category contains only one query *Propose refactorings for this class*. It computes all bad smells of the selected class and proposes refactorings for these bad smells. Figure 1 illustrates that this returns 4 results for the currently selected `OrderedCollection` class: 3 instances of the user-defined refactoring *replaceMagicNumber*, and one instance of the composite refactoring *pushUpMethodToNewClass*. This list of proposed refactorings only mentions refactorings for which the preconditions are satisfied. This is achieved by invoking the appropriate method for checking preconditions in the *Smalltalk* refactoring engine. From the list of all proposed refactorings, the user can directly execute refactorings by invoking the appropriate method in the *Smalltalk* refactoring engine. For example, in Figure 1 we see that selecting the composite refactoring *pushUpMethodToNewClass* opens a new window in which a new superclass needs to be specified, to which the method will be pushed up.

Metrics. We have also implemented a category of logic queries that compute object-oriented metrics. These can be used to detect those places in the code that are worthy of further investigation because they are likely candi-

dates for bad smells. Once these locations in the source code have been identified, we can use our other queries to analyse these parts of the program in more detail. Marinescu [8] used object-oriented metrics to identify structural weaknesses and bad smells in object-oriented class hierarchies. Simon *et al.* [11] also used metrics to identify bad smells and propose adequate refactorings.

3 Selected Bad Smells

In this section, we explain the three bad smells that we have chosen to detect during our experiments.

Unused Parameter

One of the things that needs to be detected by the speculative generality bad smell of Beck and Fowler [6] is the presence of unused method parameters. A method defines an unused parameter if it includes a formal parameter in its signature that is never used in its implementation. The implementation of a method may span several classes, of course, since the method can be overriding a method from a superclass or be overridden in a subclass. Given a particular class, the developer does not know beforehand which method implementation he has to inspect, nor does he know the highest superclass that implements the method, which subclasses of the class override which methods, and which subclasses don't. Because of this, detecting the occurrence of an unused method parameter manually is a very hard and time consuming process. It boils down to checking whether the highest definition of the method and none of its overriding methods uses this parameter. Therefore, we need to traverse the parse tree of each of these methods to find out whether or not the parameter is used.

Unused parameters can be removed by applying the *removeParameter* refactoring. This refactoring makes sure that the unused parameter is removed from a particular method, all of its overriding methods, and all of the methods callers. The refactoring requires as arguments the class in which the method is defined, the method defining the unused parameter, and the unused parameter itself.

Inappropriate Interfaces

A class defines an inappropriate interface if some of its sibling classes define an interface that is not fully supported by the class itself, nor by some of its other siblings. Good interfaces are extremely important when designing flexible and reusable object-oriented systems. Any situation in which the interface of a class is inappropriate, incomplete or unclear should thus be avoided at all costs.

Detecting the inappropriate interface bad smell manually is quite difficult because one has to analyse an entire class hierarchy, and the interfaces it defines. To automate the detection of inappropriate interfaces in a hierarchy of classes, we use the following algorithm: (1) retrieve all direct subclasses of the root class of the hierarchy; (2) compute all possible subsets of this set of classes; (3) for each of these subsets, compute the intersection of the interfaces of all classes contained in the subset; (4) in each of the resulting intersections, exclude all those methods that are present in the interface of the root class of the hierarchy. Clearly, this algorithm grows exponentially with the number of subclasses, because we compute all possible subsets. Therefore, we restrict it by only considering subsets of three or more classes that should share an interface of two or more methods.

There are two solutions to resolve the problem of inappropriate interfaces. A developer can either insert an intermediate superclass between the root class of the hierarchy and the subclasses that implement a shared interface, or he can augment the interface of the root class of the hierarchy with the interface shared by the subclasses. These two solutions correspond to an *addClass* and an *addMethod* refactoring respectively. The *addMethod* refactoring requires as arguments a list of methods to be added and the root class to which they should be added. Similarly, the *addClass* refactoring requires as arguments the root of the hierarchy, a list of subclasses of this root class that should become subclasses of the newly introduced class, and a list of selectors that are shared by the subclasses and that should be implemented in the newly introduced class.

Duplicated Methods

Duplicated methods, or duplicated code in general, seriously hamper software evolvability, since it becomes easy to oversee a method implementation when bug fixes or changes should be made. Manual detection of duplicated methods is not straightforward. The developer has to walk through each method implementation in a class hierarchy, and mutually compare each of these methods, statement by statement, to verify whether they are duplicated or not. Given the fact that methods may consist of many different statements, and may use different variable names, although this does not affect the duplication, this is a hard and time consuming task.

The algorithm we use to detect *duplicated methods* is based on a statement-by-statement comparison of the parse trees of the methods, rather than comparing their string representation. This ensures that white spaces, comments, and renamings of temporary variables can be ignored. One particular restriction we took in this article, is to apply the duplicated method smell only to siblings of a particular class. Although this is a very strict definition of duplicated methods, it already allowed us to identify some major design

flaws, as our experiments will show.

The refactorings that can be applied to remove duplicated methods depend upon the particular context in which this bad smell occurs: (1) if two or more siblings contain a duplicated method, and their common superclass does not define that method, a single *pushUpMethod* refactoring pulls up the method to the superclass and removes it from the siblings; (2) if only a small number of siblings contain a duplicated method, and their common superclass does not define that method, but the remaining siblings should not inherit the method implementation, an *addClass* refactoring should first introduce an intermediate superclass between the siblings, and afterwards a *pushUpMethod* refactoring should push up the duplicated method from the siblings to the newly introduced intermediate superclass; (3) if some siblings contain a duplicated method, and the common superclass also defines that method, an *addClass* refactoring should introduce an intermediate superclass, after which the duplicated method should be pushed up to this new class by a *pushUpMethod* refactoring.

4 Experiments

This section summarises the experiments we conducted to validate our approach. We first introduce the three case studies on which we tested our techniques, and then report upon the results we obtained for these case studies in the subsequent subsections.

4.1 Selected case studies

The first selected application is *Soul* [14], an interpreter for a logic programming language developed at our lab. Its implementation consists of 150 classes and 1966 different method implementations, which makes it a small to medium-sized application. *Soul* is a research prototype, which is being worked on by a team of developers. As such, it is under constant evolution, which makes it an ideal subject for our experiments.

The second selected application is the *Smalltalk Collection* hierarchy. The hierarchy consists of 101 classes and 1999 methods. It is an essential part of the *Smalltalk* programming environment, and as such it is heavily optimised, and not much subject to changes. It is interesting to analyse this hierarchy to find out whether there are still any remaining bad smells and opportunities for refactoring.

A final application is *HotDraw* [3], a small-scale application framework in the domain of structured drawing editors. It is a popular, successful, well-documented framework that has undergone many evolutions. As such, it is very interesting to verify whether we can still identify some design flaws among its 69 classes and 886 method implementations.

In the following subsections, we first report upon the bad smells our tool identified. Then, we discuss the refactorings that were proposed based on these bad smells, and we elaborate upon the refactorings that were effectively applied to remove the bad smells.

4.2 Unused Parameter Revisited

In the *Soul* application we detected 5 occurrences of the unused parameter bad smell. Based on these occurrences, a number of *removeParameter* refactorings were proposed. We investigated each of these refactoring opportunities to check and decided to apply all of them.

The *Collection* hierarchy contains six instances of the unused parameter bad smell. Six *removeParameter* refactorings were proposed based on these bad smells. A closer investigation of the source code revealed that they should all be applied.

In the *HotDraw* framework, the unused parameter bad smell occured 26 times. Based on these bad smells, 26 *removeParameter* refactorings were proposed. Two of the proposed refactorings were effectively applied. Because the other 24 bad smells and proposed refactorings all occurred in the same `ToolState` class, we decided to investigate this class in more detail. Analysis revealed that the class was actually defined in the wrong class hierarchy. It should be defined as a subclass of the `Controller` class, which is part of the Model-View-Controller paradigm. Indeed, `Controller` deals with exactly the same events as `ToolState`, and defines all of the methods which were reported to have an unused parameter. If `ToolState` becomes a subclass of `Controller`, all unused parameter occurrences simply disappear. Hence, the actual refactoring that needs to be applied is a *moveClass* refactoring.

4.3 Inappropriate Interface Revisited

The *Soul* application contains four inappropriate interfaces. Based on these occurrences, a combination of *addClass* and *addMethod* refactorings was proposed, that either insert an intermediate superclass or implement the method in the complete class hierarchy. We decided to apply one of the proposed *addMethod* refactorings, but the proposed *addClass* refactorings were not applied. Instead, we decided to change existing inheritance relationships between the involved classes because this resulted in an improved and cleaner design. Additionally, it also solved all reported bad smells.

In the *Collection* hierarchy we found 4 occurrences of the inappropriate interface bad smell. Based on these bad smells, 8 refactorings were proposed: 4 *addClass* refactorings that add an intermediate superclass and 4 *addMethod* refactorings that provide an implementation for the detected

methods in the entire class hierarchy. Of these 8 proposals, we decided to apply 4 proposed refactorings to restructure the *Collection hierarchy*. We searched for other refactoring opportunities after these refactorings had been applied. Some duplicated code was detected, and 14 additional *pushUpMethod* refactorings were proposed for all methods that are duplicated. Because intermediate superclasses have been introduced by the previously applied refactorings, each of these newly proposed refactorings can be executed to remove the duplication.

For the *HotDraw* framework, only one occurrence of the inappropriate interface bad smell was detected in the classes `CompositeFigure` and `ViewAdapterFigure` that have a common superclass `Figure`. Based on this bad smell, one *addClass* and one *addMethod* refactoring is proposed. Neither of these two refactorings is applied, however. A closer inspection of the involved methods reveals that they all form part of the standard *VisualWorks* framework, and that they are actually implemented by the `VisualComponent` class, which is a common superclass of both `CompositeFigure` and `ViewAdapterFigure`. Since our detection algorithm only checks the classes belonging to the *HotDraw* framework, it did not take into account methods implemented by the `VisualComponent` class. The resulting bad smell is thus actually a false positive.

4.4 Duplicated Method Revisited

The *Soul* application only contains a single occurrence of the duplicated method bad smell: the classes `QuotedCodeTerm` and `SmalltalkTerm` contain exactly the same implementation for the `printForCompileOn:` method. Moreover, these two classes have a common superclass `SymbiosisTerm`, which has no other subclasses. A *pushUpMethod* refactoring is thus proposed, that pulls up the `printForCompileOn:` method. Clearly, this refactoring should be applied to remove the code duplication and improve the structure of the application.

In the *Collection* class hierarchy, seven duplicated method bad smells were identified. Based on these bad smell occurrences, 7 refactorings were proposed. 4 of them were simple *pullUpMethod* refactorings, and 3 of them were composite refactorings consisting of the introduction of an intermediate superclass followed by a *pullUpMethod*. The intermediate superclass is introduced first since it is not possible to simply pull up the method to the common superclass, because this superclass already implements it, or because it has too many other subclasses. We decided not to apply the composite refactorings since we would be introducing a class which would define only one method. The four *pushUpMethod* refactoring proposals could be applied, however.

The *HotDraw* framework contains only two duplicated methods. First of all, `RectangleFigure` defines a method `rectangle:` and `EllipseFigure` defines a method `ellipse:` that both contain the same method body. Secondly, `RectangleFigure` and `EllipseFigure` both have exactly the same implementation for the `displayFigureOn:` method. Because `RectangleFigure` and `EllipseFigure` are subclasses of `Figure`, we can not simply pull up the methods. Both `rectangle:` and `ellipse:` are initialisation methods that do not belong to the `Figure` class, and the `displayFigureOn:` method is already defined as an abstract method in class `Figure`. Therefore, we propose to rename the `rectangle:` and `ellipse:` methods so that they share the same name, and then perform an *addClass* and *pushUpMethod* refactoring, to add an intermediate superclass between `Figure` and `RectangleFigure` and `EllipseFigure` and pull up the `displayFigureOn:` method from `RectangleFigure` and `EllipseFigure` to this new intermediate superclass. Applying this refactoring also opens the opportunity to remove the code duplication in the `displayFilledOn:` and `displayOutlineOn:` methods in the `RectangleFigure` and `EllipseFigure` classes.

4.5 Discussion

From the experiments we learned a number of things: in each of the three case studies, almost all of the bad smells that were detected pinpointed situations that were worthy of our attention. Of the 65 bad smells we identified, there were only 5 false positives.

The proposed refactorings were not always the ones that were needed. In some cases, the user needed to perform some extra analysis to identify which refactoring was necessary. We are convinced that the rules for proposing refactorings can be fine-tuned to further increase their accuracy and usefulness. For example, we could add more contextual information, such as the simultaneous occurrence of multiple bad smells in the same or related software entities.

Finally, we didn't encounter a big difference in the number of detected bad smells between the three considered case studies. Even the *Collection* hierarchy, that is considered to be very stable, contained a number of bad smells that could be removed easily.

5 Scalability

Detection of some bad smells can be very computation intensive. As such, it is not always feasible to check them on a large code base. Therefore, we could combine our approach with more lightweight approaches that optimise detection of some bad smells, at the expense of losing precision. For example, a limited subset of bad smells can be

expressed in terms of regular expressions that are more efficient to compute. In *Smalltalk VisualWorks*, we can use the *Code Critic* tool for this purpose, which is a *Lint*-like tool [7] that has been extended to include global design information. This tool can detect a number of simple bad smells, but remains limited in scope. For example, none of the bad smells used in our experiments are detectable with *Code Critic*.

To detect duplicated methods we could rely on more efficient approaches that have been reported upon in literature. For example, Ducasse *et al.* [4] sketch an approach to detect duplicated code in an (object-oriented) application based on line-based string matching. Baxter *et al.* [2] uses a more flexible, but more computation-intensive approach by relying on a ful-fledged parser. Our approach also takes the full parse tree into account, which allows us to detect code that is similar but not entirely identical. Balazinski *et al.* [1] implemented an automated refactoring tool to factorise cloned code, but our refactoring tool is more generic because it can be applied for any kind of bad smell.

Since one particular bad smell can often be remedied by a multitude of refactorings, a large list of refactorings will be proposed in practice. To keep this manageable, we can use a number of techniques:
(1) Induce an order on the proposed refactorings. For example, a *removeMethod* refactoring will be listed before a *removeParameter* refactoring for the same method, since the application of the former refactoring makes the latter one obsolete.
(2) Use user-configurable threshold values. For example, only detect bad smells (and their associated proposed refactorings) if at least three classes and/or three methods are involved.
(3) Use composite refactorings. For example, the `pushUpMethodToNewClass` refactoring shown in Figure 1 is in essence a composite refactoring that includes the primitive refactorings *addClass* and *pushUpMethod*.
(4) Use a visualisation mechanism. This is for example proposed by van Emden and Moonen [13] who combine the detection of bad smells in *Java* with a visualisation mechanism.
(5) Use metrics. Metrics are another way to address scalability, and have been investigated in the context of bad smells and refactorings in [11, 8].

6 Conclusion

In this paper, we proposed an advanced refactoring tool that also provides support for the earlier phases of the refactoring process, by detecting bad code smells in a software application, and proposing refactorings that could be applied to remove these smells. We integrated our tool in the *Smalltalk* development environment, and integrated it with the already available *Refactoring Browser*. The tool was validated by detecting three different bad smells for three different case studies on medium-sized object-oriented applications. All detected bad smells pinpointed situations that were worthy of our attention, and many of the proposed refactorings were actually useful to resolve the bad smells.

References

[1] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Advanced clone analysis to support object-oriented system refactoring. In *Proceedings 7th Working Conf. Reverse Engineering*, pages 98–107. IEEE Computer Society Press, 2000.

[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detecting using abstract syntax trees. In *Proc. Int'l Conf. Software Maintenance*, pages 368–377. IEEE Computer Society Press, 1998.

[3] J. M. Brant. Hotdraw. Master's thesis, University of Illinois at Urbana Champaign, 1995.

[4] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. Int'l Conf. Software Maintenance*, pages 109–118. IEEE Computer Society Press, September 1999.

[5] R. Fanta and V. Rajlich. Reengineering Object-Oriented Code. In *Proc. Int. Conf. on Software Maintenance*, pages 238–246. IEEE Computer Society Press, March 1998.

[6] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[7] S. Johnson. Lint, a C Program Checker, 1978.

[8] R. Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, Politehnica University of Timisoara, 2002.

[9] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana Champaign, 1992.

[10] D. Roberts, J. Brant, and R. Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[11] F. Simon, F. Steinbrückner, and C. Lewerentz. Metrics Based Refactoring. In *Proc. 5th European Conference on Software Maintenance and Reengineering*, pages 30–38. IEEE Computer Society Press, 2001.

[12] L. Tokuda and D. S. Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, 8(1):89–120, 2001.

[13] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proc. 9th Working Conference on Reverse Engineering*. IEEE Computer Society Press, October 2002.

[14] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 2001.