Refactoring Tools and Complementary Techniques

Martin Drozdz[1], Derrick G Kourie[2], Bruce W Watson[2], Andrew Boake[3]
Espresso Research Group, Department of Computer Science
University of Pretoria, Pretoria, South Africa 0001
[1]martind@eject.co.za, [2]{dkourie, watson}@cs.up.ac.za, [3]andrew@systemiclogic.com

Abstract

Poorly designed software systems are difficult to understand and maintain. Modifying code in one place could lead to unwanted repercussions elsewhere due to high coupling. Adding new features can cause further quality degradation to the code if proper design and architectural concerns were not implemented. Development in a large enterprise system with such attributes will, over time, lead to a myriad of concerns unless the system is periodically overhauled or refactored in some way.

Refactoring can aid the developer to improve the design of the code and to make it cleaner, without changing its behaviour. This study provides answers for some of the questions on refactoring. A refactoring tool survey is given. The IDEs surveyed include some of the most popular commercial and open source offerings from IntelliJ's IDEA, IBM's Eclipse and Sun's Netbeans. We also explain a way to automatically find targets for refactorings via automatic detection of code smells from static code analysis. Concerns on viewing compiler refactorings as a fully automated refactorings are raised. We will perform a critical evaluation of refactoring by surveying these tools.

1. Introduction

This section contains answers to the what, why and where questions pertaining to refactoring.

1.1. What is refactoring?

One definition of the word "factor" means to influence something. To refactor means to re-influence something that already exists. There is however a more precise meaning in the computer science context. In order to successfully refactor a program, one needs to change (and thus re-influence) the program in such a way as to improve the design and simultaneously preserve its behaviour.

1.2. Why refactor?

One needs to know how refactoring fits into the software engineering process in order to see how one can benefit from it.

Continuous design [Shore 2004], which utilizes refactoring, allows one to add more flexibility into the design, by adding to an initially simple design as the need arises, instead of having a big upfront design. Thus the design will evolve as the code grows. There is a shift from building software towards growing it. The process of refactoring can be used to contribute to these evolving states of the code.

[Garlan 1994] mentions that: "Object-oriented systems also have some disadvantages. The most significant is that in order for one object to interact with another (via procedure call) it must know the identity of that other object. This is in contrast, for example, to pipe and filter systems, where filters do need not know what other filters are in the system in order to interact with them. The significance of this is that whenever the identity of an object changes it is necessary to modify all other objects that explicitly invoke it. "

Automated refactorings manage to soften the disadvantages of the object orientated architectural style, by not only modifying the part of the object that is invoked, but all of the invokers as well. Code dependencies are seamlessly resolved as the refactoring tool will usually utilize an abstract syntax tree (AST) or code database. The AST or code database will hold the crucial code dependencies that need to be updated whenever a refactoring is made.

1.3. Where to refactor?

Refactoring can be used in different contexts. For example: to improve the design and code quality of existing systems; to help evolve the design of systems dynamically through incremental development with practises such as test driven development and agile

methodologies, thereby negating the need for a big upfront design; to understand how existing code works [Fowler 1999]; to manage change in a software organisation [Beck 2000]; to perform refactoring at an architectural level [Van Kempen 2005]; and to refactor code, with design patterns as targets of the refactoring [Kerievsky 2004];

1.4. Recent Research

[Mens 2004] provides a thorough look at research in the domain of software refactoring and software restructuring. In the survey it is mentioned that although commercial refactoring tools have begun to proliferate, research into software restructuring and refactoring continues to be very active, and remains essential to reveal and address the shortcomings of these tools. The following tool survey takes a closer look at such refactoring tools. A critical analysis is done in order to evaluate the IDEs that implement these tools and how well they are integrated into the IDE.

1.5. Summary

A continuous design process helps one to evolve a design so as to make the maintenance phase less costly by adding more flexibility and reusability into the code. Continuous design can also be used to solve any design issues overlooked previously. Refactoring plays an important role in continuous design and certainly has benefits, but the process must be adequately managed to ensure that it is done properly. Various refactoring tools are available. A well-chosen refactoring tool can not only facilitate the refactoring task, but can also support the management of the refactoring endeavour. The next section gives a tool survey of the most prominent Java-based refactoring tools.

2. Tool Survey

This is the main part of our paper in which we discuss the tool survey performed on various IDEs with refactoring tool support. This research was conducted in both an academic and commercial environment during the first half of 2005 and includes motivations for Java as the language of choice, the IDEs selected for the survey as well as a detailed analysis of the IDEs.

2.1. Language Choice

This article sees Java as the preferred language for refactoring, mainly due to its open source nature, its popularity and the amount of refactoring support it has gained in the form of books, articles and automated tools available for it.

Refactorings have spread into many different languages, but Java remains the language with the most refactoring tool support, with C# lagging somewhat behind. A list of recent refactoring tools is available for several languages [Fowler 2005].

2.2. Survey Motivation

The aim of this tool survey was to evaluate the refactoring tool support in common Open Source and commercial IDEs. For the purpose of this survey we chose IBM's Eclipse, IntelliJ's IDEA and Suns Netbeans.

There is currently a choice between manual and automated refactorings. Refactorings tools allow one to automate refactorings. The IDE can further automate this by searching for targets of refactorings by using either static analysis or a compiler approach which will be introduced later.

These issues can influence the ease of refactoring greatly. We want to find practical, inexpensive solutions for enterprises that wish to evolve their designs through refactoring.

The automated refactorings in most modern IDEs allow one to preview the resulting impact the refactorings will have on the code. The preview also shows warnings when code could be negatively impacted. The automated refactorings take the entire project code under consideration. This saves one a lot of time compared to manual refactorings which are error prone and rely on the compilation process to highlight errors caused by unchecked dependencies or the like.

2.3. Findings

Netbeans 4.0 was severely lacking in its refactoring abilities. Even with the recent Jackpot project headed by James Gosling which was meant to boost such features, Netbeans only supports 4 refactorings at the moment. It lacks even the extract method refactoring which is used as a benchmark for refactoring support.

Eclipse 3.1 and IDEA 4.5 both support around 30 similar refactorings. IDEA 5.0 however, is far ahead of Eclipse in terms of ease of use and productivity of refactoring.

Although the number of refactorings that an IDE supports is important, there is one aspect in which IDEA is the clear winner. IDEA has an array of static code analysis tools (almost 500 in IDEA 5.0) which enable one to pick up hotspots pertaining to abstraction and encapsulation issues, method and class metrics, class structure, performance issues and many more. Some of these issues map directly to code smells [Fowler 1999].

The latter are, of course, candidates for refactorings. Fowler [1999] explains code smells in detail and also provides several refactorings to eradicate these smells.

From the nearly 500 static code analysis techniques in IDEA we managed to sift out 10 code smell checks. The more familiar we became with code smells; the easier it was to map the static code checks in IDEA to code smell checks. Code smells require one or more refactorings in order to be removed. Some IDEs do not support all of the refactorings, which means that manual refactorings will have to be attempted instead.

Manual refactorings are error-prone as the developer cannot automatically detect the many dependencies that need to be fixed when a refactoring is made. For example take the 'Change method signature' refactoring which can remove or add a parameter in a method over an entire project. A developer will manually have to search for places where the method is called and change the methods signature in each place, while automatic refactoring does this work for him, more reliably and accurately. The same concept goes for renaming a package or class. Why do it manually in 20+ classes when the refactoring tool can do it with one click?

Our analysis was done on two Java J2EE financial trading projects that are currently in production. IDEA 4.5 was used to analyse the code base. IDEA 4.5 has many categories in which the static analysis checks are grouped into in order to allow one to focus on specific problem areas. For example, one will find a category called abstractions issues with 14 static analysis checks. As a proof of concept the static analysis checks that where of most interest to us in this category where the ones pertaining to the following 3 checks that we individually selected and which could be mapped to code smells.

1. Feature Envy – When a method calls other methods from another object or class more than 3 times, it suggests that some functionality of the caller should rather be in the class being called. The methods in this code smell context would be either class or object instance methods.

2. Magic Number – This refers to a literal value that is used directly, not through the use of a constant variable. This did not seem very serious at first, but we often found this to be a very irritating code smell. Magic Numbers should be linked to classes as constants and declared as public static final variables. This ensures that other classes could make use of the variables also ensures that the Magic Number can be associated with a meaningful variable name which describes its purpose more clearly. Magic Numbers thus lead to much confusion and are a very serious code smell, in large projects, where they can cause many errors.

3. Switch Statement – A switch statement is very easy to detect, even without a fancy tool. All that is needed is to search for the string literal "switch" in code. Generally the lack of switch statements is a sign of good object orientated code, since most switch statements should have been replaced with polymorphism.

Code smells recognised by [Fowler 1999] are not necessarily the only code smells in existence, but it is not possible to mention all the possible code smells as new smells emerge very quickly as tool support increases for refactoring.

Some unrecognised code smells already have legitimate refactorings. For example IDEA can search for class fields that have public or package scope. These are candidates for the refactoring 'encapsulate field'. There is a check for 'if statements' with too many branches, which can be refactored using 'Replace nested conditional with guard clauses'.

One particularly useful check is that of the search for redundant throws clauses. These are particularly irritating code smells, as they can introduce unnecessary try-catch blocks and confuse the intent of the code. Both IDEA and Eclipse can eradicate redundant 'throws' clauses, albeit in different ways. The two IDEs supply automatic refactorings for this code smell.

In IDEA and Eclipse, the automatic detection of these code smells is also very often followed by the option of performing an automatic refactoring. The developer will see a list of the line numbers linked to the code smells occurring in specific classes. Eclipse will highlight the code smells in yellow and normally provide an automated refactoring suggestion by a clickable icon on the left hand side of the code. IDEA provides a synopsis of the refactorings or techniques available to eradicate the code smell. The developer needs to select the refactoring proposed by the refactoring tool to execute it. Eclipse's detection mechanisms are done through its compiler, while IDEA relies on static code analysis.

These features allow one to further automate the removal of code smells, without having to look for them for hours or to know the needed refactorings by heart. If a developer is provided with knowledge of how to map the code smells with the corresponding analysis checks then most of the hard work is already done.

Knowing all of the code smells and resulting refactorings to apply can be a problem, if these mappings are not provided, especially when one considers developers who are trying to learn how to refactor. The fact that there are over 20 basic code smells, 30 refactorings possible in the IDE, and over 70 refactorings in total to learn makes the learning curve for refactoring rather steep.

Even though the refactorings in Eclipse and IDEA are automated, Fowler [1999] still suggests that automatic refactorings need to be performed sequentially and unit tested accordingly. There are however fully automated code smell detections which are done by compilers and

thus deemed to be exempt from the need to re-test after a refactoring is performed. This is not the case in Eclipse's compiler, which still needs human intervention to ensure correctness, even with the knowledge that the compiler performed the code smell checking and should therefore have picked a code smell which could be refactored without errors. We explain why this is so in the following paragraphs.

Developer intervention in refactoring is vital as [Mens 2004] points out; fully automated refactoring tools can make the code look worse after the refactoring has taken place, by doing too many refactorings. Compiler refactorings seem rather trivial at first when one considers that they have to do more with unread local method variables or unread private class member variables or methods, which can be removed without causing any impact on the surrounding code. If setup correctly Eclipse can automatically detect and highlight these code smells as soon as one opens a class. These smells are normally deeply hidden and hard to weed out. Eclipse can quickly remove these smells without much developer intervention. One can just click on the code smell warning icon and a solution to code smell is provided and performed with one click of the mouse button. The developer is able to setup the compiler environment so as to take note or ignore these code smells when compiling a project or dynamically compiling a class upon entry.

In large projects these code smells are rather serious, especially when a large class with big methods has been edited by several developers, who unknowingly left behind several unused variables and methods. These code smells make the code far harder to understand and less readable, increasingly polluting the code, and after a couple of years can cause severe maintenance problems. The project we where busy with during our research had code that was 4 years old and mostly the large classes which where used most often by all developers were affected by this type of code smell. With a CVS tool one is able to safely delete this type of code and retrieve it from previous versions if needed.

The concern we have with Eclipse's approach and thus automatic compiler refactorings in general is in the case where unused variables are initialized via constructors or methods. The code executed in the constructors or methods themselves could have unwanted side effects even if the variable, to which the class instance or primitive value is assigned, is not used. For example lets assume that in "*int i = x();*", variable *i* is not used and the method *x()* has side effects. If the unused variable *i* is deleted along with the method *x()*, then this will cause a change in behavior if the rest of the program depends on the side effects of *x()*.

Such side effects are bad programming in the first place, but we cannot be assured that they do not exist when refactoring unused code. Therefore there are still cases where human intervention is required and even compilers cannot free us from having to test refactorings to ensure their required behavior preserving properties. We can therefore conclude that compiler code smell checks can still result in behavior changes once a refactoring removes the smell. There is no reason why such behavior changes cannot occur in fully automated refactoring approaches as described in [Mens 2004].

A simple approach to ensure behavior preservation with compiler refactoring is to eliminate the unused variables but to ensure that the method and constructor logic is still called. Otherwise careful analysis of the constructor and/or method code is required to protect against unwanted side-affects.

3. Conclusion and future work

We intend to continue to delve into research topics that make refactoring easier to accomplish. We can see from the above tool survey that there are very good tools which help one accomplish refactoring rather painlessly. We are also investigating contributing to the refactoring tool support of Eclipse. We will do this by developing a refactoring plug-in or smell-detection framework for Eclipse.

4. References

[Garlan 1994] GARLAND, D, SHAW, M: *An Introduction to Software Architecture*, Carnegie Mellon University

[Fowler 1999] FOWLER, MARTIN. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.

[Beck 2000], Beck, K, Fowler, M: *Planning Extreme Programming*, Addison Wesley

[Kerievsky 2004] KERIEVSKY, JOSHUA: *Refactoring to Patterns*, Addison Wesley, http://www.industriallogic.com/xp/refactoring/index.html, accessed 2005-08-15

[Mens 2004], MENS, T and TOURWÉ, T: *A Survey of Software Refactoring, IEEE Transactions on software engineering*, VOL. XX, NO. Y, MONTH 2004

[Shore 2004], JIM, SHORE: *Continuous Design*, http://martinfowler.com/ieeeSoftware/continuousDesign.pdf, accessed 2005-08-11

[Fowler 2005] FOWLER, MARTIN: *A list of refactoring tools for several languages,* http://www.refactoring.com/tools.html, accessed 2005-08-13

[Van Kempen 2005] VAN KEMPEN, MARC: *Studies into Refactoring of Software Architectures*, Technishe Universiteit Eindhoven, Masters Thesis