Proceedings of the 7th WSEAS International Conference on Automation & Information, Cavtat, Croatia, June 13-15, 2006 (pp138-143)

A Framework for an Adaptive Refactoring Tool

CAPT. ALAIN COUTU, CD, MSC
Aerospace and Telecommunications Engineering Support Squadron
Canadian Forces Bases
Trenton, Ontario
CANADA

CATHARINA SERINO, PHD
School of Business
North Carolina Central University
Durham, North Carolina
USA

SUZANNE SMITH, PHD
Computer and Information Sciences Department
East Tennessee State University
Johnson City, Tennessee
USA

SARA STOECKLIN, PHD
Computer Science Department
Florida State University
Panama City, Florida
USA

Abstract - Refactoring is the process of making changes to the internal structure of existing code without changing the external behavior of that code. The resulting code is more flexible, reusable, and maintainable. While refactoring is becoming more popular in the software development community, manual refactoring can be a long and tedious process. Tools that support refactoring are becoming available; however, many provide only limited types of refactorings and require heavy user intervention. This paper presents an open source framework for an adaptive refactoring tool. The framework allows easy addition of new refactorings or modification of existing ones. An implementation of the framework is described in this paper.

Key Words- Software Tools and Environments, Software Engineering, Refactoring, and Design Patterns

1 Introduction

Refactoring is the process used by software engineers to transform existing code into a more reusable and adaptable structure while keeping the integrity of the code's functional requirements. The purpose of performing refactoring is to increase program maintainability, flexibility and understandability. Other benefits of refactoring include making it easier to add new code, improving the design of existing code, gaining a better understanding of code, and making the code less annoying [1]. Such refactoring also allows designers to experiment with new design ideas [2].

Refactoring is not a new idea. Smalltalk programmers have been performing refactoring manually since the inception of the language [3]. Structured programmers have used the techniques of cohesion and coupling to refactor code since the 1970's [4]. Object orientation and extreme programming [5, 6, 7, 8] have been driving forces in increasing the use and popularity of refactoring in recent years.

Many refactoring tools [9, 10, 11, 12] currently exist. Martin Fowler [10] provides a comprehensive list of these tools with descriptions and links to the tools' perspective websites. Most of these tools provide a limited number of refactorings. Some of the most commonly implemented refactorings include:

- Extract method, field or class
- Rename parameter, field, method, or class
- Encapsulate field
- Push down and up field or method
- Remove method, field and class

These tools also vary greatly in their implementation. Some tools implement a refactoring with automated code generation, some require heavy human intervention, and others only highlight code under suspicion.

While these tools help with the long and tedious process of manual refactoring, they do have limitations. The requirement for heavy human intervention is one such limitation. However, as mentioned by Roberts et al. [3], Tokuda and Batory [2], and Riggs and Stoecklin [13], total automation of refactorings without any human intervention may not be possible. With certain refactorings, there exist situations where a developer needs to provide information (e.g., variable or method names) or accept suggested refactoring transformations. Other limitations in current refactoring tools include their inability to recognize the need for patterns, see commonality of code, and allow users to define needed refactoring methods. Despite the limitations of current refactoring tools, the need to automate the refactoring process is widely recognized in the software development community.

2 Tool Overview

The software discussed in this paper is an implementation of an innovative open-source framework for a refactoring tool. The framework is based on a generic notion of the refactoring process. Extracting the generic notion of refactoring requires the separation of the characteristics of the refactoring process from characteristics of a particular refactoring. This separation is accomplished through the use of design patterns. The framework for this refactoring tool is constructed using design patterns to implement polymorphic behavior. This current software engineering practice provides an adaptive software architecture [14]. The goal of such an architecture is to build a system that it is easily modified to add new or tailored refactorings. Such adaptability allows software developers to modify the refactoring tool to meet their specific industry coding standards or domain-specific software engineering needs.

Adaptive Refactoring Tool, ART, is the implementation of this framework. It implements several refactorings in order to demonstrate the potential of this framework. The framework structure of ART provides software developers with an easily maintainable, independent implementation of each refactoring. It also allows adaptability to refactor code written in various languages. ART performs refactoring transformations on code written in any language with a Backus Naur Form (BNF) description.

The generic underlying refactoring process used by ART is the following:

1. The user initializes ART parameters to define specifics such as refactorings that are to be automatically implemented without human intervention, refactorings to ignore, and parameter constraints of refactorings.
2. The user selects the source code file(s) for refactoring and specifies the choice of language.
3. ART reviews (parses) the code and identifies potential refactorings.
4. ART implements the refactorings identified in step one as automated without human intervention.
5. ART highlights other potential refactorings and displays the original code in the left

display area as shown in Figure 1 at the end of the paper.

6. The user reviews the highlighted code, selects a refactoring to implement, and inputs any needed data such as a variable name. Default names are provided by ART if user inputs are not available.
7. ART modifies the code to implement the refactoring, compiles the code to assure syntactic correctness, and displays the refactored code in the right display area as shown in Figure 1.

3 Tool Description

ART consists of four major components which are:

a) parser (any BNF-described languages),
b) refactoring transformer,
c) deparser, and
d) graphical user interface (GUI).

The parser reviews the source code selected by the user. Using the appropriate language grammar, the parser extracts the tokens. When being refactored, new tokens are created for the various refactoring transformation strategies. These tokens are all of the same class type (i.e., AbstractToken) but may be of different types. The AbstractFactory pattern creates these tokens by the refactoring strategies without having the knowledge of the type of tokens being created. The parser component of ART has been created using the JavaCC tool, freely available at http://www.suntest.com/JavaCC. JavaCC permits the generation of language parsers based on grammars for an object-oriented language such as Java and C++.

The refactoring transformer of ART is responsible for modifying the set of tokens provided by the parser to identify potential refactorings. The user is able to specify some refactorings to be implemented automatically by ART. ART also allows other refactorings to be selected through human intervention. In implementing these refactoring transformations, ART groups the refactored code in a package to provide access to all the classes of the package.

An example of a refactoring transformation provided in ART is Replace Magic Number with Symbolic Constant. In this transformation, ART searches the vector of tokens for any occurrence of a numeric value being assigned to a variable or used in an arithmetic operation. When such an occurrence is found, ART replaces the numeric value with a constant variable. The constant variable name can be specified by the user or named by ART.

Another example of a refactoring transformation provided in ART is Encapsulate Fields. ART searches the vector of tokens for the declaration of a variable. When a non-private occurrence is found, two methods, a set and a get, are created with the appropriate parameters. The vector is then searched for occurrences of the variable being used or being assigned a value. When found, the set or get method, with the appropriate parameters, replaces the variable name usage in the program to ensure proper encapsulation.

After the refactoring transformer completes the refactoring, it compiles the refactored code in ART (i.e., without having to exit the application). The compilation is implemented using the package sun.tools.javac.Main. This package returns either true or false based on the results of the compilation.

In the refactoring transformer component of ART, all transformations are implemented using the Strategy pattern. The RefactoringStrategy superclass holds the subclasses for each type of refactoring. Each independent refactoring strategy method contains the set of rules for implementing a specifc refactoring and is polymorphically selected to execute. Adding new refactorings requires only the adding of another strategy subclass as an extension of the RefactoringStrategy superclass and modification of the GUI to allow individual selection of that strategy. Modification of existing refactorings requires only overriding methods with newly modified methods or, if needed, changing the open source. Figure 2, shown at the end of the paper, shows an abbreviated class diagram for ART. The RefactoringContext, RefactoringStrategy, EncapsulateFields, and ReplaceMagicNumber classes were created for the implementation of the Strategy Pattern. The algorithms used for performing the refactoring transformations were tailored using the description of the refactoring found in [6]. The TokenFactory and AbstractFactory classes were created to generate new tokens which are added to the code vector.

After refactoring is completed, the parsed source code is viewed in the ART environment using the deparser component of ART, as shown in Figure 1. The deparser component also displays the resulting refactored code. Like the refactoring transformation component, the deparser uses the Strategy pattern to select the appropriate language formatting.

4 Adaptive Framework Design

The adaptive framework design of ART is accomplished using several design patterns. Design patterns provide solutions to common programming problems. These solutions are expressed as collaborations between classes. The general benefits of using design patterns include the provision of a mechanism to develop highly cohesive modules with minimal coupling. The patterns also isolate the variability that occurs in problem domains. This variability occurs in ART with the various refactoring transformation rules. ART allows easy modification of the various refactoring rules allowing adaption to different programming environments and local standards. This allows easy addition or modification of ART to allow the tailoring of the refactoring methodology to a particular problem domain.

While many design patterns are used in ART, the AbstractFactory pattern and Strategy pattern illustrate adaptivity of ART's framework. Grand [15] defines those forces (i.e., considerations) needed for the use of the AbstractFactory pattern as: "a system that works with multiple products should function independently of the specific product that it is working with, should be possible to configure a system to work with one or multiple members of a family of products, class instances intended for interfacing with a product should be used together and only with that product, remaining of the system work with a product without being aware of specific classes used for interfacing with the product and system should be extensible to work with additional products by adding additional sets of classes". The AbstractFactory pattern is used in the implementation of step 3 of the ART refactoring process. In this step, the source code is parsed into tokens. When being refactored, new tokens are created for the various refactoring transformation strategies. These tokens are all of the same superclass type, AbstractToken, but may be of different subclass types. By using the AbstractFactory pattern, the tokens are created by the refactoring strategies without having the knowledge of the particular type of tokens being created.

Grand [15] defines the forces of implementing the Strategy pattern as: "a program has to provide multiple variations of an algorithm or behavior, behavior variations can be encapsulated in various classes providing a consistent access methodology to these behaviors, and classes using these behaviors do not require knowledge of the implementation of the behaviors". In ART, there are two situations lending themselves to the use of the Strategy pattern. When performing steps 4 and 5 of the ART refactoring process, tokens are reformatted. This behavior varies based on the type of token. When reformatting, the deparser only needs to know that a token is being reformatted. The Strategy pattern determines the strategy for the specific reformatting.

The other situation lending itself to the Strategy pattern is the selection of the refactoring transformations. Each of the transformations consists of a set of rules that must be followed. The Strategy pattern is implemented as each refactoring is a different strategy. This common behavior, needed for all refactorings, is placed in the strategy superclass. Only the behavior of individual transformation rules is included in the subclasses.

5 Conclusion

The need for tools and techniques to increase program maintainability, flexibility and understandability is well recognized by the software development community. With refactoring becoming more practiced and new refactorings continuously emerging, automating the refactoring process is becoming critical for the wider acceptance and practice of this design technique. ART, the open-source tool described in this paper, not only automates the refactoring process but also provides a mechanism to easily add new refactorings and modify existing ones. ART is not completely implemented but is intended to prove the framework concept and to demonstrate the strong potential of the automation of the refactoring process.

References:

[1] J. Kerievsky, *Refactoring to Patterns*, Addison-Wesley, 2004.

[2] L. Tokuda and D. Batory, Evolving Object-oriented Designs with Refactorings, *Journal of Automated Software Engineering,*, Vol. 8, 2001, pp. 89-120.

[3] D. Roberts, J. Brant, and R. Johnson, A Refactoring Tool for Smalltalk, *Theory and Practice of Object Systems Special Issue: Object-Oriented Software Evolution and Re-Engineering*, Vol. 3, No. 4, 1997, pp. 253-263.

[4] C. Gane and T. Sarson, *Structured Systems Analysis:Tools and Techniques*, Prentice-Hall, 1979.

[5] Beck, K., Extreme Programming Explained: Embrace Change. Addison-Wesley, 2000.

[6] W.F. Opdyke, Refactoring, Reuse and Reality, *Lucent Technologies/Bell Labs website*, 1999.

[7] D. Roberts, *Practical Analysis for Refactoring*, Ph.D. Thesis, University of Illinois, Urbana-Champaign, IL, 1999.

[8] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[9] L. Huiqing, C. Reinke, and S. Thompson, Tool Support for Refactoring Functional Programs, *Proceedings of the ACM SIGPLAN workshop on Haskell*, Uppsala, Sweden, 2003, pp. 27-38.

[10] http://www.refactoring.com.

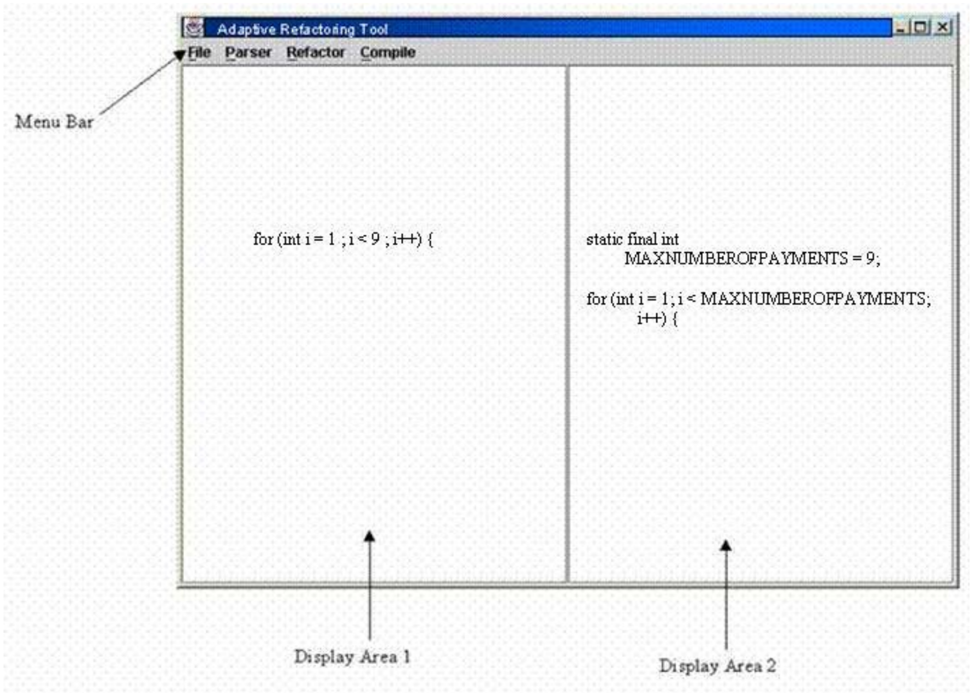
[11] E. Glynn Mealy and P. Strooper, Evaluating Software Refactoring Tool Support, *Proceedings of the Australian Software Engineering Conference (ASWEC)*, Sydney, Australia, 2006, pp. 1-10.

[12] J. Simmonds and T. Mens, A Comparison of Software Refactoring Tools, *Technical Report*, 2002.

[13] R. Riggs and S. Stoecklin, Automated Process for Code Refactoring, *Proceedings of The 8th International Conference on Information System Analysis and Synthesis*, Orlando, FL, 2002.

[14] K.L. Lieberherr, *Adaptive Object-oriented Software: the Demeter Method with Propogation Patterns*, PWS Publishing Company, 1996.

[15] Grand, M.. *Patterns in Java, Volume 1, A Catalog of Reusable Design Patterns Illustrated with UML*, John Wiley, 1998.



```
for (int i = 1 ; i < 9 ; i++) {
```

```
static final int
      MAXNUMBEROFPAYMENTS = 9;

for (int i = 1; i < MAXNUMBEROFPAYMENTS;
        i++) {
```

Fig. 1 ART Environment

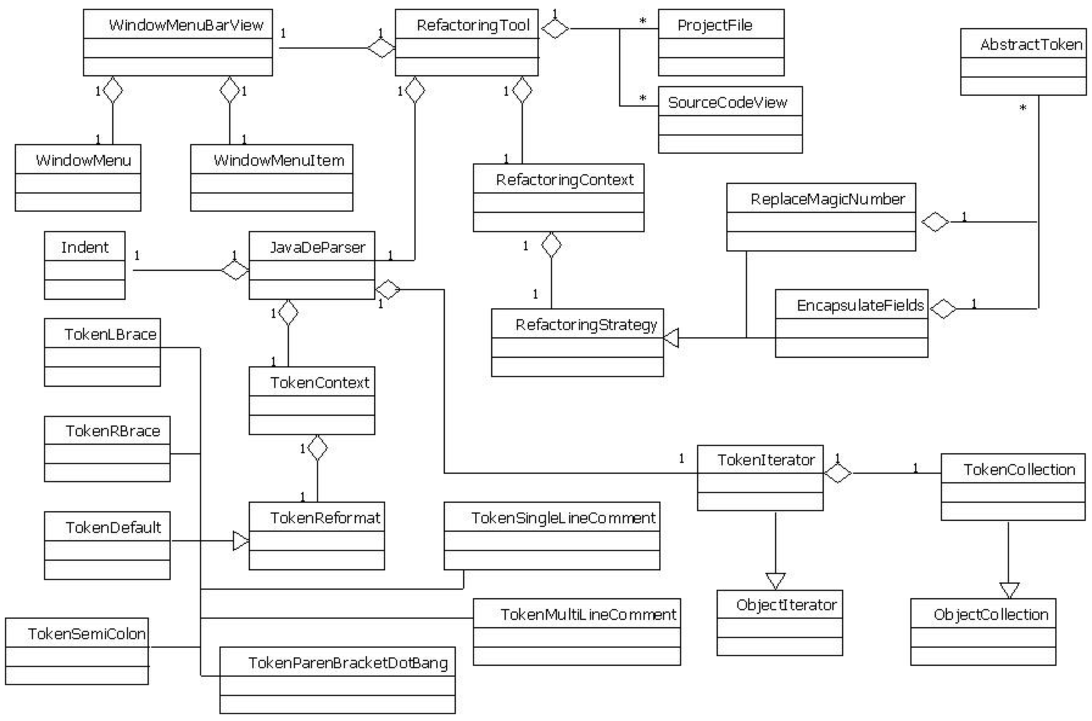


Fig. 2 Class Diagram for ART