Introduction of Aspect Oriented Techniques for refactoring legacy software

Dr S.A.M.Rizvi, Jamia Millia Islamia, New Delhi
Zeba Khanam, JSS Academy of Technical Education, Noida

Abstract
Refactoring has become a well-known technique for improving the code in a way that preserves behavior. The application of refactorings during development process of an object oriented or procedure oriented software improves the design and therefore the quality of software. During the evolution of software it is a requirement to refactor them in order to make it more compatible and flexible with the new environment. Much work is being done in refactoring object oriented code with aspect oriented programming. But this paper describes the various types of refactoring being done on procedural codes for eg: C language and the utility of refactoring the procedural codes with the help of aspect oriented programming. The paper also proposes certain refactorings that could be achieved in a better way using AOP.

Keywords: Legacy Systems, Aspect Oriented Programming, refactoring techniques, Object Oriented Refactorings, procedural languages challenges.

1. Introduction

Refactorings may be applied manually, although manual code manipulation is error prone and cumbersome, so maintainers need tools to make automatic refactorings. Refactoring is being done since long time back especially in procedural languages as there are many features like exception handling, logging concerns etc that are not handled very well by these languages .Its essence is applying a series of small behavior-preserving transformations, each of which "too small to be worth doing"[3].Refactoring can be done manually as well as automatically. Currently extensive literature on refactoring object-oriented programs and some very good tools for refactoring Smalltalk and Java code are available. The feature of aspect oriented programming has helped a lot in refactoring the software as it helps modularize the program in a better way. The C programming language, especially the preprocessor directives that coexist with it, complicates refactorings in different ways as directives are not legal C code and may not support correct refactorings. Refactoring C poses two major research challenges. On the one hand, as preprocessor directives may violate correctness, new precondition and execution rules must be defined for existing refactorings to preserve behavior. On the other hand, the automated execution of refactorings requires specialized program analysis tools to represent and manipulate preprocessor directives [2].

2. Background: Refactorings for C

Various research attempts were made in the area of refactoring C.The very first attempt was made by W.Opdyke [4] in his own PhD thesis .It was established as a prime contribution in refactoring techniques developed and catalogued to help the maintainers with manual process.

Certain explorations are done in the area of refactoring the C language [1]. A catalogue of refactorings was proposed for the C language that was implemented in a prototype tool [1]. Later on it was discovered that this catalogue was not applicable to the preprocessor directives used in C language. The refactorings allowed on preprocessed C code is very restricted; else it could happen that the directives become irrecoverable which could change the structure of the complete source code and its behavior too, hence violating the primary rule of refactoring.

Thus refactoring Code with preprocessor directive requires that users should be able to transform preprocessor directives; e.g., adding a parameter to a macro definition; - the presence of directives should not affect the correctness of refactorings; users should be able to transform C code that has interleaving preprocessor directives. Especially in preprocessor directives it is difficult to refactor the macros and conditional statements. The other concern is that of code entangled in long procedure bodies with interleaved switch cases that also causes problem in refactoring. Thus these problems need to be resolved with the emerging refactoring techniques to help a software evolve in a better way.

3. Problems in refactoring procedural codes

This section highlights the problem in refactoring the procedural code.

3.1Problem with Macro definitions: Refactoring Macros

When refactoring C code, the macro code that is called in that scope has to be taken into consideration. Macro has access to global variables and modifies them globally, the macro definitions tend to change with any change in their variables. Furthermore, the studies show that correctness of refactorings can be affected due to the following reasons [2].

-if a macro is defined but never called in the scope of refactoring;

- if a macro refers to a variable with different declarations, and the macro is called from the different contexts of the variable;
- If a macro definition uses the concatenation operator ##.

Previous works have proposed a list of refactorings for macro definitions.

-Rename Macro
-Rename Macro parameter
-Add parameter to macro definition
-Remove parameter from macro definition
-Add macro definition replacing value in code
-Remove macro definition

3.2Conditional Directives

Conditional directives also pose a challenge to refactoring. Conditional directives lines are those starting with #if, #ifdef, #ifndef and #elif, plus #else lines and #endif lines. There are no refactoring tools that can deal correctly with conditional directives. The usual practice is to deal with the preprocessed code ,but that would discard some of the code mentioned under the conditional directive ,so refactoring would be done on only part of the code which would lead to a lot of discrepancy in the code.

Some of the refactorings proposed in earlier works, for the conditional directives are stated as:

-Eliminate an alternative
-Complete a statement inside a conditional branch with the code that follows the conditional
- Move common code outside the conditional

"Refactoring C is difficult", the standard C scanner, parser and AST builder no longer apply when directives are not preprocessed, as the code does not respond to the C grammar [2].Thus we can conclude that macros and conditional compilation cause error in refactoring and pose problems for refactoring. Some of the solutions too have been offered [2]. New refactorings for preprocessor directives, new definitions of scope, additional preconditions and execution rules for existing refactorings have been proposed.

3.3 Problem with long procedures: Refactoring procedures

In a traditional C implementation, the major complexity is created by union and a procedure body that is essentially a giant switch statement. In C++ implementations, the switch statement is distributed over classes representing the cases, this made the code more modular and reduced maintenance effort, while speeding up the interpreter. Thus, subclassing, and reducing the conditional statements, may improve both the clarity of the design and the run-time performance. Now this subclassing can be achieved in object oriented languages but for the procedural languages, this again is a big drawback. So , how can the aspects improve upon these code complexities, and can it be used in subclassing and simplifying conditionals has to be validated and is the focus of this research paper.

The objective of this research paper is to highlight the problems being posed during the time of C language refactorings and to explore as to how aspect oriented refactorings helps in refactoring the C code and what contributions can be done by the aspect oriented refactorings in refactoring the above mentioned problems.

4. Aspect Oriented Refactorings

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*)* [5]. 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. Aspect oriented refactoring has been used for code clone removal and has performed better than object oriented refactorings.

Systems software uses conditional compilation to manage crosscutting concerns in a very fine-grained and efficient way, but at the expense of tangled and scattered conditional code. Refactoring of conditional compilation into aspects gets rid of these issues, but it is not clear yet for which patterns of conditional compilation aspects make sense and whether or not current aspect technology is able to express these patterns [6].

AOP does provide more type safety and more power than usual macros. But how it will be carried out is work in progress. In the next section we will discuss how refactorings are done in Object oriented framework. Then the next section states how we can make the AOP also work to refactor on legacy applications using these guidelines.

5. Refactoring Object Oriented Framework

Some of the refactorings that can be done on object oriented systems are described below. The paper proposes here that these refactoring methods can be extended to the legacy languages also with the help of aspect oriented approach. Several techniques have been developed based on structured programming guidelines these include goto elimination, case statement refinement and other techniques.

1. Refactoring To Generalize: Creating an Abstract Superclass
2. Refactoring To Specialize: Subclassing and Simplifying Conditionals
3. Capturing Aggregations and Reusable Components.
4. Moving Members between Aggregate and Component Classes.
5. Converting an Association, Modeled Using Inheritance, Into an Aggregation.

Supporting Refactorings

1. Creating a Program Entity:
2. Deleting a Program Entity:
3. Changing a Program Entity:
4. Moving a Member Variable:
5. Convert a code segment to a function.

6. Aspect-Oriented Programming

Aspect-Oriented Programming, or AOP, extends Object-Oriented Programming with the concept of aspects, which modularize crosscutting concerns. Like a class, an aspect is intended to capture a set of related program elements addressing a particular concern. Like a class, an aspect is intended to capture a set of related program elements addressing a particular concern. Unlike classes, however, aspects are intended to modularize crosscutting concerns—those that inherently span the definitions of many classes. AOP techniques let the programmer specify well-defined ways that aspect code blends with other program code.
Many software systems must address concerns that are not localized to a single class.AOP has contributed in solving many such problems. For example:

6.1. Error handling: Legacy applications like C applications do not have an explicit exception handling mechanism .Instead they typically rely on an idiomatic approach for signaling and handling exceptions. One common idiomatic approach is the “return code idiom” , in which a special return code signals an exception has occurred. Michael Mortensen has described an aspect-oriented approach for throwing exceptions in place of the “return code idiom”, and discusses using aspects to handle those exceptions in a modular way [7].

6.2Code clone removal: Code clones are identical codes repeated across the code base, are special kind of code smells and can be seen as homogenous crosscuts.AOP refactoring have worked well in handling this problem[9].

These examples illustrate crosscutting concerns, and modularizing the crosscutting concerns even with best object oriented techniques is not possible, thereafter making the refactoring procedure even tougher.

7. AOP refactoring statically and dynamically

In order to allow aspects to modify classes and their hierarchy, aspects may include several forms of introduction, which declares new members on classes (inter-type declaration) or alters inheritance relationships between classes.

For dynamic changes in the program execution the joinpoint model specifies which join points in the program execution can be described. Based on joinpoint specifications, code contained in an aspect can be invoked during execution and affect behavior at runtime.

The code that specifies how program behavior is to be affected at runtime is advice. Advice has a great deal of power to inspect program state at runtime using reflection, and to manipulate state and execution paths. An Advice may get executed

before a joinpoint: advice can view and modify input values and other state before the joinpoint is entered.
after a joinpoint: advice can view and modify return values and other state after a joinpoint has finished. There are also special cases of after advice for methods returning normally or exiting by throwing an exception.
around advice replaces the joinpoint.

8 .Refactoring procedural languages with AOP

In the following we present the features of AOP that can be used in refactoring the procedural code and propose the guidelines to refactor the code and thus can be used to improve the modularity and maintainability of the system.

8.1 Aspects versus classes

Since with the emergence of aspect oriented languages, the utilities of aspects can very well be added to the procedural languages also as the aspect code can be very well blended with the procedural code. The experiments are performed using AspeCt Oriented C (ACC) on the source code in C.ACC provides a compiler that translates code written in ACC into ANSI-C code. This code can be compiled by any ANSI-C compliant compiler, like for example gcc. The refactorings mentioned in the next section are being performed on C using ACC. The current work is being done on refactoring macros. In the previous section we had explored the problems caused by macros, we intend to replace them with the appropriate addition of aspects and advice and observe the refactoring changes on the code. In the next section few of the AOP refactorings are discussed those are being used to refactor the source code.

Refactoring aspect oriented software [8] has explored few of the contributions that AOP makes in the field of refactoring. We are employing some of these refactorings to the procedural code.

Aspects behave quite similar to classes .So the refactorings that can be done on object oriented applications can be done on procedural code also as most of the refactorings done on classes can also be carried using aspects. Following are the refactorings that we propose using aspects.

1. Refactoring to Generalize: **Creating a superaspect**-One of the refactoring in OOPs to separate the design is to move the common behavior of a set of concrete class to an abstract superclass. In case of procedural code the code entangled in different functions with the same behavioral characteristics can be migrated to the superaspect and be used whenever required. This would generalize the common behavior into a single entity.

2. Refactoring to Specialize: subclassing using aspects- Some times it is required that a complex function is transformed into a subclass. Migrating the code from a complex function to sublass reduces the complexity of that function. Refactoring through subclassing cannot be achieved in a language like C but with the help of AOP introduction of sub aspects can serve the purpose by moving the fields to the sub aspects.

3. Refactoring by Creating a Program Entity-As in OOPs a new program entity like class can be created or a new variable can be added but this class is unreferenced. No, instances of this class are created nor any subclasses are created, so the behavior is preserved. Similarly using, AOP we can create an
1. Create Empty Aspect
2. Create Named Pointcut
3. Create Empty Advice

4. Refactoring by Deletion of a Program Entity- The unreferenced variable can be deleted and it also preserves behavior as they are unreferenced .This can be achieved using deleting unreferenced Introduced field and deleting set of unreferenced introduced methods.

5. Refactoring by Moving Program Elements-Certain member variables are migrated to subclass or superclass, this can also achieved in procedural languages by:
1. Moving an advice declaration from one aspect to another.
2. Moving member variables to Aspects and from aspects to functions.

6. Refactoring by replacing the macro definition codes with proper aspect program and advice-This is another refactoring that we propose. The macros cause a lot of problems in refactoring as the refactoring tools cannot be applied to them as we had discussed in the earlier sections. Thus we are looking for appropriate replacements of the macros that would also preserve the behavior. The replacements can be done using appropriate aspects and executing the advice on the appropriate join points. But to what extent would this refactoring help, has to be validated.

We have discussed some of the refactorings that can be performed on aspect oriented software. Infact the conventional refactoring done using OOPs can be reinvented using AOP and that too with the more enhanced features of aspect and join-point model.

9. Conclusion – In this paper, we presented our idea of guidelines for refactoring procedural code by using aspect oriented programming. Therefore, we made a classification of refactorings which adds the aspect join-point feature to the legacy procedural code. Earlier these refactorings or features could not be added to the procedural code because they were basically applied to classes and the languages that have the utility of classes. These refactorings we are employing on the procedural language like C, for introducing subclasses, superclasses, moving larger functions to aspects, turning the code of macros into the code encapsulated in aspects and many more, to increase the readability and reduce the complexity of the software to help it evolve in a better way. Work in this area is still not being explored too much so automatic refactoring tools are not available for all kinds of refactoring. Currently, we are working on these techniques and implementing them using ACC (Aspect Oriented C) and we intend to explore other techniques also that could assist in the same. Our emphasis is mainly on manual refactoring techniques and work is still in progress. The introduction of these refactoring to legacy languages has given a new direction to the old conventional refactoring techniques as well as a new shape to the old source code with even enhanced capabilities.

References

[1] Garrido, A. Software Refactoring Applied to C Programming Language. MS Thesis. University of Illinoisate Urbana-Champaign, 2000.
[2]Garrido.A, Ralph Johnson "Challenges of Refactoring C Programs", 2002
[3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1st edition, 1999.
[4] William Opdyke, "Refactoring object Oriented Frameworks", 1992.
[5] Jan Hannemann, "Aspect-Oriented Refactoring: Classification and Challenges", 2005

[6] Wolfgang De Meuter, Bram Adams, "Can we Refactor Conditional Compilation into Aspects, 2008
[7] Magiel Bruntink, Arie van Deursen, "Discovering Faults in Idiom Based Exception Handling", 2005
[8] Shimon Rura, "Refactoring Aspect-Oriented Software", 2003.
[9] S.Schulze, M.Kuhlemann, M Rosenmuller "Towards a Refactoring guideline Using Code Clone Classification", 2008.