Chapter 85
Resolving Conflict in Code Refactoring

Lakhwinder Kaur
Apeejay Institute of Management Technical Campus, India

Kuljit Kaur
Guru Nanak Dev University, India

Ashu Gupta
Apeejay Institute of Management Technical Campus, India

ABSTRACT

Refactoring is a process that attempts to enhance software code quality by using small transforming functions and modifying the structure of the program through slightly different algorithm. It is important to analyze the design pattern of the software code as well as the impact and possibility of the application of some conflicting refactorings on it. The objective of this chapter is to present an approach for analyzing software design patterns in order to avoid the conflict in application of available refactoring techniques. This chapter discusses the mechanism to study software code or design patterns to automate the process of applying available refactorings while addressing the problem of conflict in their application.

INTRODUCTION

The term 'Refactoring', which was first introduced by William Opdyke and Ralph Johnson, and then popularized by M. Fowler, is meant to improve internal code structure without altering external functionality of a software system. It attempts to enhance code quality by using small transforming functions and modifying the structure of the program through slightly different algorithm. After pattern analysis in programs, software code quality can be enhanced by minor modifications in the program design and that is the stage where Software Refactoring comes into play. It is observed that developers can either streamline the size of a software product after development and simplify its design, or speed up its download and execution speed with the help of Refactorings. This fundamental activity also aims to help organizations in maintaining and updating their software

DOI: 10.4018/978-1-4666-4301-7.ch085

more easily. It is an iterative process and it must result in significant improvement over original version of the software.

In fact, Refactoring facilitates reverse engineering process also, which is a process of analyzing and extracting patterns in source code programs in order to understand their system functionality and representation. As the software needs to be continuously monitored and updated in order to meet the rapid changes in requirements of the industry, there is a crucial need to exploit reverse engineering during different stages of the software development. Program design analysis is required in reverse engineering in order to ensure consistency of the software components.

In brief, when applied in a disciplined way, the summed up impact of refactoring must be significant for improving the efficiency and quality of code. But the problem is that automated, assisted or even manual application of refactorings suggested so far, may result in either negative or lesser impact on the quality of software than it was targeted. The reason is- some pairs of refactoring techniques are opposite to each other and they may nullify each other's impact when performed in a particular order. Hence, it is important to analyze the design pattern of the software code as well as the impact and possibility of the application of some conflicting refactorings on it. This chapter attempts to analyze code design patterns, which may lead to conflict in the application of refactorings and suggests a mechanism to avoid it.

BACKGROUND

Though the work of Opdyke (1992), and Johnson(1997), has been the pillar behind introducing refactoring techniques, the work by M.Fowler(1999) suggested various refactorings. Since then, a number of researchers are engaged in finding effective ways of applying refactorings in software codes. Refactoring techniques are applied at various levels starting from design level to various control structures in a program. R. Najjar, S. Counsell, G. Loizou, and K.Mannock(2003) proved that by replacing constructors with factory methods and making minor modifications to the interface provided by the class, the lines of code can be reduced and classes can provide better abstraction. They have shown the positive effect of refactoring on softwares by improving its quality and efficiency. Mens and Tourwe(2004), performed a comprehensive survey of the research in refactoring upto that time. They classified research according to Five criteria and presented that a tool or formal model for refactoring should be sufficiently abstract to be applicable to different programming languages, but it should also provide the necessary interface to add language-specific behaviour.

Tokuda and Batory(1995) proposed automated search for refactoring trends. They implemented 12 object oriented database refactorings described by Banerjee and Kim(1987). They were able to automate thousands of lines of changes with a general-purpose set of refactorings. They implemented the refactorings in C++ and expressed the difficulty in managing C++ preprocessor Information. Beck(2000) suggested that there is correlation between characteristics of the rapid software development and the need for frequent refactorings. In case of rapid software development, programmers do not bother about code simplicity, understandability and maintainability. M. Boger et al(2002) introduced refactoring browser integrated in a UML modeling tool. They discussed how Refactorings can be extended to static architecture as well as to dynamic behaviour.

Moving state and behavior between classes can help reduce coupling and increase cohesion(Pressman,2001). The cumulative effect of several simple refactoring steps and the available tool support for their automated application has made the refactoring process a widely accepted technique for improving software design. However, identifying the places where refactoring should be applied is neither trivial