Chapter 8.15
Heuristics and Metrics for OO Refactoring:
A Consolidation and Appraisal of Current Issues

Steve Counsell
Brunel University, UK

Youssef Hassoun
University of London, UK

Deepak Advani
University of London, UK

ABSTRACT

Refactoring, as a software engineering discipline, has emerged over recent years to become an important aspect of maintaining software. Refactoring refers to the restructuring of software according to specific mechanics and principles. While in theory there is no doubt of the benefits of refactoring in terms of reduced complexity and increased comprehensibility of software, there are numerous empirical aspects of refactoring which have yet to be addressed and many research questions which remain unanswered. In this chapter, we look at some of the issues which determine when to refactor (i.e., the heuristics of refactoring) and, from a metrics perspective, open issues with measuring the refactoring process. We thus point to emerging trends in the refactoring arena, some of the problems, controversies, and future challenges the refactoring community faces. We hence investigate future ideas and research potential in this area.

INTRODUCTION

One of the key software engineering disciplines to emerge over recent years is that of refactoring

(Foote & Opdyke, 1995; Fowler, 1999; Hitz & Montazeri, 1996; Opdyke, 1992). Broadly speaking, refactoring can be defined as a change made to software in order to improve its structure. The potential benefits of undertaking refactoring include reduced complexity and increased comprehensibility of the code. Improved comprehensibility makes maintenance of that software relatively easy and thus provides both short-term and long-term benefits. In the seminal text on the area, Fowler (1999) suggests that the process of refactoring is the reversal of software decay and, in this sense, any refactoring effort is worthwhile. Ironically, Fowler also suggests that one reason why developers do not tend to undertake refactoring is because the perceived benefits are too "long term." Despite the attention that refactoring has recently received, a number of open refactoring issues have yet to be tackled and, as such, are open research concerns. In this chapter, we look at refactoring from two perspectives.

This first perspective relates to the heuristics by which refactoring decisions can be made. Given that a software system is in need of restructuring effort (i.e., it is showing signs of deteriorating reliability), IS project staff are faced with a number of competing choices. To illustrate the dilemma, consider the question of whether completion of a large number of small refactorings is more beneficial than completion of a small number of large refactorings. A good example of the former type of refactoring would be a simple "rename method," where the name of a method is changed to makes its purpose more obvious. This type of refactoring is easily done. An example of the latter, more involved refactoring, would be an "extract class" refactoring where a single class is divided to become two. This type of refactoring may be more problematic because of the dependencies of the original class features.

As well as the decision as to "what" to refactor, we also look at the equally important decision as to "when" we should refactor. Throughout all of our analysis, we need to bear in mind that refactoring offers only a very small subset of the possible changes a system may undergo at any point in its lifetime. We return to this theme later on.

Combined with the need to choose refactorings and the timing of those refactorings, the need to be able to measure the refactoring process is also important. Software metrics (Fenton, 1996) provide a mechanism by which this can be achieved. A metric can defined as any quantifiable or qualitative value assigned to an attribute of a software artefact. The second perspective thus relates to the type of metric applicable for determining firstly, whether a refactoring is feasible, which of competing refactorings are most beneficial and how the effects of carrying out refactoring have impacted on the software *after* it has been completed. In terms of "when" to refactor, a metrics program implemented by an organization may provide information on the most appropriate timing of certain refactorings according to metric indicators as, for example, a rapid and unexplained rise in change requests.

For both perspectives investigated, there are a large number of issues which could possibly influence their role in the refactoring process. For example, most refactorings can at best only be achieved through a semi-automated process. For example, the decision on how to split one class into two can only be made by a developer (and aided by tool support once that decision has been made). Some metrics are subject to certain threats to their validity and are thus are largely inappropriate for judging the effect of a refactoring; the lines of code (LOC) metric is a good example of such a metric because of the unclear definition of exactly what a line of code is (Rosenberg, 1997). In our analysis, we need to consider these issues.

The objectives of the chapter are three-fold. Firstly, to highlight the current open issues in the refactoring field. In particular, some of the associated problems that may hamper or influence