ISSN No. 0976-5697

Volume 2, No. 4, July-August 2011

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Making Your Code Agile with Refactoring: An Agile Software Development Technology

Saumya Goyal*
Computer Science and Engineering Department
UIET, Kurukshetra University,
Kurukshetra , Haryana, India
er.saumya.goyal@gmail.com

Sona malhotra
Computer Science and Engineering Department
UIET, Kurukshetra University,
Kurukshetra , Haryana, India
sona_malhotra@yahoo.com

Abstract: This paper provides an extensive overview of existing research in the field of software refactoring. This research is compared and discussed based on a number of different criteria: the refactoring activities that are supported, the specific techniques and formalisms that are used for supporting these activities, the types of software artifacts that are being refactored, the important issues that need to be taken into account when building refactoring tool support, and the effect of refactoring on the software process.

***Keywords*:** Agile Software Development, eXtreme Programming, Refactoring Process, Refactoring Activities, Applications Example.

I. INTRODUCTION

An intrinsic property of software in a real-world environment is its need to evolve. As the software is enhanced, modified, and adapted to new requirements, the code becomes more complex and drifts away from its original design, thereby lowering the quality of the software. Better software development methods and tools do not solve this problem because their increased capacity is used to implement more new requirements within the same time frame, making the software more complex again. To cope with this spiral of complexity, there is an urgent need for techniques that reduce software complexity by incrementally improving the internal software quality. Refactoring is basically “the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure” [1]. The key idea here is to redistribute classes, variables, and methods across the class hierarchy in order to facilitate future adaptations and extensions. In the context of software evolution, refactoring is used to improve the quality of the software (e.g., extensibility, modularity, reusability, complexity, maintainability, efficiency).

II. PROCESS SUPPORT

Refactoring is an important activity in the software development process. It is the process of changing a software system aiming at organizing the design of the source code, making the system easier to change and less error-prone, while preserving observable behavior. This concept has become popular in agile software methodologies, such as Extreme Programming (XP), which maintains source code as the only relevant software artifact. Although refactoring was originally conceived to deal with source code changes, the concept can be extended to include similar transformations on structural models of the system. In this section, we discuss how refactoring fits into the process of agile software development.

A. Agile Software Development

Typically, major reengineering efforts are carried out only when the software has already degraded so much that it has turned into legacy code. In contrast, the agile software development community, with eXtreme Programming (XP) as its main proponent, suggests supporting a culture of continuous reengineering. They propose a process where one develops and reengineers software in small iterations: You develop a little (to implement the desired behavior), reengineer a little (to improve the structure), develop a little more, and so on. Unfortunately, these short iterative development cycles do not seem to fit very well in a more classical software development process.

Refactoring is one of the cornerstones in the XP process. Many object-oriented IDEs provide considerable support for XP, using a combination of refactoring support and unit testing, two core activities in XP. Van Deursen et al. show that refactoring of test code is different from refactoring production code in two ways [2]:

A. There is a distinct set of bad smells involved and

B. Improving test code involves additional test-specific refactoring.

III. REFACTORING

Program restructuring is a technique for rewriting software that may be useful either for legacy software as well as for the production of new systems. The internal structure is changed, although the behavior (what the program is supposed to do) is maintained. Re-structuring reorganizes the logical structure of source code in order to improve specific attributes, or to make it less error-prone when future changes are introduced.

In the context of object-oriented development, behavior-preserving program changes are known as refactorings. The refactoring concept was introduced by Opdyke, yet becoming popular by Fowler's work and Extreme Programming (XP)[2], an agile software development methodology. According to Fowler, refactoring is the process of changing a software system in such a way that it does not alter the observable behavior of the source code, yet improving its internal structure. In this context, a refactoring is usually composed of a set of small and atomic refactorings (the mechanics), after which the target source code is better than the original with respect to particular quality attributes, such as readability and modularity.

Refactoring can be viewed as a technique for software evolution throughout software development and maintenance [5]. Software evolution can be classified into the following types:

a. Corrective evolution: correction of errors;
b. Adaptive evolution: modifications to accommodate requirement changes;
c. Perfective evolution: modifications to enhance existing features.

A. Extreme Programming and Refactoring

One of the main reasons for the wide acceptance of refactoring as a design improvement technique is its adoption by Extreme Programming (XP) [3]. In XP, refactorings are applied in specific parts of the code that contain inconsistencies (“code smells"), and unit tests check the software output after structural changes. In particular, besides managing software evolution, refactoring is intrinsic to each XP development activity [4]. XP practices guide simple implementation according to immediate user requirements, promoting successive refactorings for improving design before adding new features.

B. Refactoring Principles

Principles for the process of refactoring are as follows [3]:-

a. Reduce code
b. Avoid clever code – keep it simple
c. Make it small and cohesive – single responsibility
d. Eliminate duplication
e. Eliminate dependencies – rather than striving to reduce dependencies, strive to remove them
f. Write self documenting code – make comments unnecessary
g. Code should be understandable in seconds – it is not just about reducing the amount of code, but also about clearly expressing meaning.
h. Avoid primitive obsession – focus on creating higher level abstractions
i. Check in frequently, take small steps – every commit should be only one change.

i. Shorter feedback cycle
ii. Other developers are kept in the loop
iii. Avoid large painful merges

j. Keep code at one level of abstraction – each method should do one thing, and delegate to other methods that each do one thing.

C. Refactoring Activities

The refactoring process consists of a number of distinct activities [2]:

a. Identify where the software should be refactored.
b. Determine which refactoring(s) should be applied to the identified places.
c. Guarantee that the applied refactoring preserves behavior.
d. Apply the refactoring.
e. Assess the effect of the refactoring on quality characteristics of the software (e.g., complexity, understandability, maintainability) or the process (e.g., productivity, cost, effort).
f. Maintain the consistency between the refactored program code and other software artifacts (such as documentation, design documents, requirements specifications, tests, etc.).

a) Identifying where to Apply which Refactorings

A first decision that needs to be made here is to determine the appropriate level of abstraction to apply the refactoring.

b) Guaranteeing that the Refactoring Preserves Software Behavior

By definition, a refactoring should not alter the behavior of the software. The original definition of behavior preservation as suggested by Opdyke states that, for the same set of input values, the resulting set of output values should be the same before and after the refactoring. Opdyke suggests ensuring this particular notion of behavior preservation by specifying refactoring preconditions.

In many application domains, requiring the preservation of input-output behavior is insufficient since many other aspects of the behavior may be relevant as well [7]. This implies that we need a wider range of definitions of behavior that may or may not be preserved by a refactoring, depending on domain-specific or even user specific concerns:

i. For real-time software, an essential aspect of the behavior is the execution time of certain (sequences of) operations. In other words, refactorings should preserve all kinds of temporal constraints.
ii. For embedded software, memory constraints and power consumption are also important aspects of the behavior that may need to be preserved by a refactoring.
iii. For safety-critical software, there are concrete notions of safety (e.g., liveness) that need to be preserved bya refactoring.

c) Assessing the Effect of Refactoring on Quality

For any piece of software, we can specify its external quality attributes (such as robustness, extensibility, reusability, performance). Refactorings can be classified according to which of these quality attributes they affect. This allows us to improve the quality of software by applying the relevant refactorings at the right places [8]. To achieve this, each refactoring has to be analyzed according to its particular purpose and effect. Some refactorings remove code redundancy, some raise the level of abstraction, some enhance the reusability, and so on. This effect can be estimated to a certain extent by expressing the refactorings in terms of the internal quality attributes they affect (such as size, complexity, coupling, and cohesion).

An important software quality characteristic that can be affected by refactoring is performance. It is a common misconception that improving the program structure has a negative effect on the program performance. In the context of logic and functional programs, restructuring transformations typically have the goal of improving program performance while preserving the program semantics. In the context of object-oriented programs, Demeyer investigated the effect of refactorings that replace conditional logic by polymorphism.

He concludes that the program performance gets better after the refactoring because of the efficient way in which current compiler technology optimizes polymorphic methods [5].

To measure or estimate the impact of a refactoring on quality characteristics, many different techniques can be used. Examples include, but are not limited to, software metrics, empirical measurements, controlled experiments, and statistical techniques.

d) *Maintaining Consistency of Refactored Software*

Typically, software development involves a wide range of software artifacts such as requirements specifications, software architectures, design models, source code, documentation, test suites, and so on. If we refactor any of these software artifacts, we need mechanisms to maintain their consistency. Since the activity of inconsistency management Bottoni et al. [6] propose maintaining consistency between the program and design models by describing refactoring as coordinated graph transformation schemes. These schemes have to be instantiated according to the specific code modification and applied to the design models affected by the change. Within the same level of abstraction, there is also a need to maintain consistency. For example, if we want to refactor source code, we have to ensure that the corresponding unit tests are kept consistent [7].

IV. APPLYING REFACTORING

A. When not to Refactor

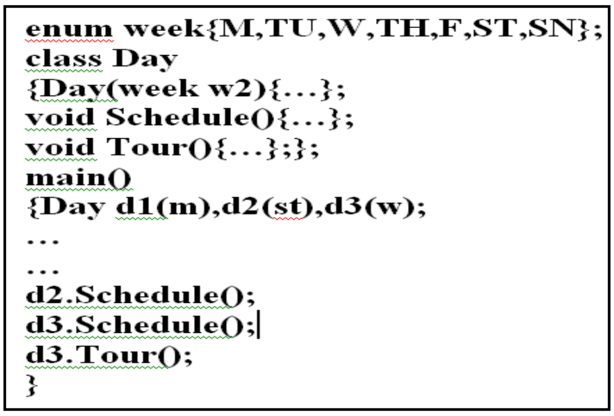
a. Never refactor while making any other code change. Take a note of the refactoring and complete it after the bug fix or functionality change has been committed [4]. Never commit a refactoring at the same time as any other code change. If you do break a build, having small, single change commits, will make spotting the problem that much easier [9].
   i. When you spot a code smell – take a note of it
   ii. Finish your change
   iii. Commit
   iv. Refactor
b. Never refactor alone. You should always have a second set of eyes on the problem. Pair programming is essential while refactoring. The second person will be their to help ensure we are making the code easier to understand, using good names, and helping to ensure we are not breaking any existing logic [10].

B. When to Refactor

a. You can refactor before or after a bug fix or a functionality change
b. If you think the change will improve the design of the code
c. If you think the change will improve the readability of the code for other developers

V. REFACTORING EXAMPLE

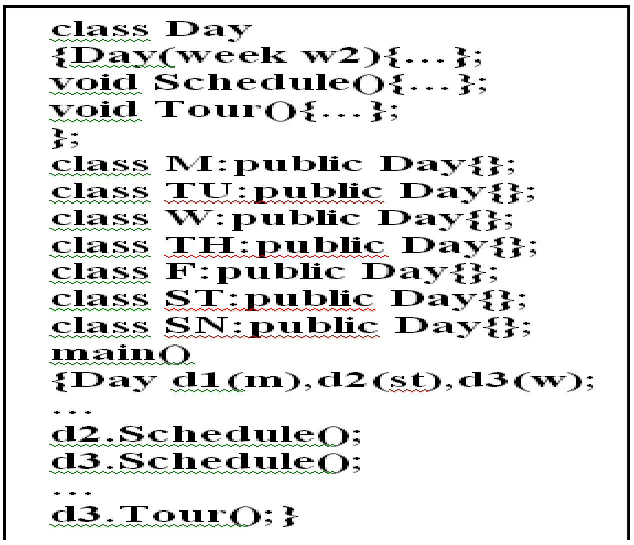
This example describes how refactorings can be implemented to improve the design of a program and hence making it faster. The purpose of this example is to clearly demonstrate how refactorings work. Let us take an example of a simple C++ program code which displays the schedule of a week.



```
enum week{M,TU,W,TH,F,ST,SN};
class Day
{Day(week w2){...};
void Schedule(){...};
void Tour(){...};};
main()
{Day d1(m),d2(st),d3(w);
...
...
d2.Schedule();
d3.Schedule();
d3.Tour();
}
```

Figure 1. Program code before refactoring.

After applying Create_empty_class refactoring which uses the values of enum type to create and name subclasses, the code becomes as follows:



```
class Day
{Day(week w2){...};
void Schedule(){...};
void Tour(){...};
};
class M:public Day{};
class TU:public Day{};
class W:public Day{};
class TH:public Day{};
class F:public Day{};
class ST:public Day{};
class SN:public Day{};
main()
{Day d1(m),d2(st),d3(w);
...
d2.Schedule();
d3.Schedule();
...
d3.Tour();}
```

Figure 2. Program code after refactoring.

VI. CONCLUSIONS

This paper describes how a simple process of agile software development can lead to make an existing code faster by applying a set of program restructuring operations (refactoring) specific to supporting reuse and faster execution of object-oriented application. Although it requires behavior preservation at every step which should be done carefully .

VII. REFERENCES

[1] Martin Erwig and Deling Ren. “Monadification of functional programs.”, Science of Computer Programming, 52(1-3):101–129, 2004.

[2] Martin Fowler. “Refactoring: Improving the Design of Existing Code.” Object Technology Series. Addison-Wesley, 2000.

[3] Simon Thompson and Claus Reinke.
“A Case Study in Refactoring Functional Programs.”, In Brazilian Symposium on Programming Languages, 2003.

[4] Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren. : “A Formal Specification for the Haskell 98 Module System.” In ACM Sigplan Haskell Workshop,2002.

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

[6] D. Sands, Bottoni, “Total correctness by local improvement in the transformation of functional programs,” Trans. Programming Languages and Systems, vol. 18, no. 2, pp. 175–234, March 1996, ACM.

[7] L. Larsen, M.J. Harrold, “Slicing object-oriented software”, in: Proc. 18th International Conference on Software Engineering, pp. 495–505, 1996.

[8] J. Mendling, H.A. Reijers, J. Recker, “Activity labeling in process modeling: empirical insights and recommendations”, Information Systems 35 (4) 467–482, 2010.

[9] H. Leopold, S. Smirnov, J. Mendling, “Refactoring of activity labels in business process models”, in: 15th International Conference on Applications of Natural Language to Information Systems (NLDB 2010), 2010.

[10] M. Minor, A. Tartakovski, D. Schmalen, R. Bergmann, “Agile workflow technology and case-based change reuse for long-term processes”, International Journal of Intelligent Information Technologies 4 (1), 80–98, 2008.