Formalization of Architectural Refactorings

Syed Muhammad Ali Shah
School of Engineering and Advanced Technology
Massey University, Palmerston North, New Zealand
m.a.shah@massey.ac.nz
+64-2102400753
PhD Enrollment Date: 01/Sep/2008

***Keywords*-architectural smells, refactoring, dependency graph, graph transformation**

I. Introduction

Refactoring refers to a software transformation that preserves the external behavior but improves the internal software structure [1]. It helps in achieving quality attributes such as modularity, maintainability, openness and evolvability in a software system. Classical refactoring focuses on code level transformations in order to get rid of unwanted structures known as code smells. A code smell is an indication of a deeper problem in the source code e.g. duplicate code, long methods, large class. There exist some tools to detect these code smells e.g. PMD, FindBugs etc. (Semi-) automated tool support to refactoring code smells is available in most of the software development environments (IDEs). For example, the Eclipse IDE provides semi-automated support for many low-level code refactorings, as described by Fowler [1]. This eases the process to perform refactoring quickly and safely.

There exist another class of refactorings that can change high-level system design and architecture [2]. Architectural refactoring refers to semantic preserving transformation of the software design, which improves the architecture quality and preserves the external system behavior [3]. The goal of architectural refactoring is to improve the quality of software architecture by improving simplicity, expressiveness, flexibility, modularity and maintainability. It is applied to the high-level design elements such as subsystems, modules, components, containers, packages, and relationships between these elements in order to get rid of architecture smells. An architecture smell is a problem in the higher level of abstraction, e.g. cycles between containers and namespaces, layering violations (referring to lower level layers directly) and so on. Architectural refactoring has an impact on different parts of the system and refactoring at the architecture level can influence design and implementation. Many of the concepts from code refactoring can be applied to the software architecture refactoring. The impact of refactoring on the architecture level can be very useful by leading to the architectural stability of the system.

A number of architectural analysis tools have been developed to identify the architecture level problems e.g. Sotograph, Lattix, Structure101, GQL4JUNG [4] etc. These tools identify the refactoring opportunities in existing projects and find architectural smells such as circular dependency between software elements, layering violations (referring to lower-level layers directly), complexity (e.g. too many classes in a subsystem) etc. None of these tool actually perform the automated refactoring. We can automate the architectural refactoring process, when we are able to formalize these refactorings. The existing specification of architectural refactorings, provided by Roock and Lippert in their book on large refactorings [5], is not precise enough to be used in the process of automation with the tool support.

In this decade several frameworks have been introduced that facilitate building systems with all the desirable properties, such as maintenance, scalability, openness, evolvability, modularity etc. These frameworks include OSGi and its derivatives, Spring Dynamic Modules and several Java Specification Requests (JSRs) such as JSR(277,291,294). In order to take advantage of these modularization techniques, many software vendors are refactoring their existing monolithic architectures to modular architectures e.g. BEA (Weblogic), IBM(Websphere), JBOSS are refactoring their architectures to adopt OSGi [6]. Another Project named as "Project Jigsaw" is initiated to refactor the Java Development Kit (JDK) into a modular architecture to improve some performance metrics such as download size, starting time and memory management [7]. These are the examples of large systems and manual refactoring is an undesirable option.

This project aims at formalizing a partial set of refactorings focusing on modularity of the system, as proposed in [3], [5], [8]. Formalization is the first step to achieve automation of the architectural refactorings, the need for which is strongly supported in [1], [3], [5], [9]. Some of the example refactorings that affect modularity are, cycles in namespaces, cycles in containers, degenerated inheritance etc [8].

In order to formalize the architectural refactoring process, we propose to develop a declarative language for refactoring. The purpose of this declarative language will be to precisely define the mechanics of the architectural refactorings. A common way to approach this is to use the divide-and-conquer strategy i.e. split a large refactoring into a number of smaller primitive refactorings. This is known as composition of refactorings and followed by Kerievsky [10]. We will follow this strategy due to the fact that any architectural refactoring drills down to code level. However, the architectural refactorings are not only limited to the code level refactorings (e.g. Abstract Syntax Tree (AST) manipulation), but may also involve other artifacts such as build scripts, deployment descriptors, configuration files etc.

We propose to use the fitness functions in terms of anti-patterns count and object-oriented metrics to ensure that refactoring process has improved the design and architecture. An important property of any refactoring is that it should not introduce new behavior or faults in the system. Therefore a refactoring requires that all program invariants hold after the transformation process. For this project we propose to evaluate our success based on refactoring a set of existing systems available in Qualitas Corpus [11]. Type checking (through compilation) and test cases are proposed to check the correctness of refactoring on the dataset, because these are the only accessible and available means in the dataset that invariants are not violated.

II. Problem Definition

Software architecture is one of the earliest design artifacts in the software development life cycle. It is composed of the architectural structure (e.g. components and interfaces), architecture style, relationship among these and other low level design artifacts such as coding, testing etc. Most of the systems start with a very clean software architecture, but with the passage of time things start getting messed due to the unstructured modifications in software. Stal [3] calls this "design erosion", where workarounds and patches in the system make software architecture difficult to maintain and evolve. However, the architectural refactoring can keep software architecture clean, easy to maintain and easy to evolve.

Most of the literature in the field of refactoring is focused on the code refactorings but the importance of architectural refactoring is also mentioned by several authors. Opdyke [12] and Roberts [13] call these refactorings as high-level refactorings. According to these authors smaller refactorings can be composed together to achieve the high-level refactorings. Fowler [1] in his book on refactoring calls these refactorings as big refactorings. According to Fowler and Kent Beck we can not get benefits from the code level refactorings unless we perform big refactorings. Roock and Lippert [5] in their book focus on large refactorings. They introduced a set of architectural smells or problems found in the high-level system design and provide a formal specification of these smells. Bourqin and Keller name architectural refactorings as high-impact refactorings. They use architecture violations as starting point for refactoring and assess the impact of refactorings through code metrics.

The major difference between code and architectural refactoring is that of scope of applicability. Code refactoring has local impact while architectural refactoring has non-local impact on the whole system. Sufficient tool support is available for code level refactorings, however there is a critical need for automation of the architectural refactorings with tool support. A need for architectural refactoring tool support is indicated in the literature [1], [2], [5], [9].

From economic perspective the architecture refactorings result in a resource demanding activity, when performed manually [1]. This is due to the fact that these refactorings are composed of so many smaller refactorings and can not be performed altogether. Therefore there exists some risk in performing these big refactorings. Architectural refactorings can be automated with the help of tool support if there is sufficient formalism available for these refactorings. Formalization of the code refactoring is widely discussed in the literature while there are a few references for architectural refactoring. As a matter of the fact software architecture refactoring is not very precise and concise when compared with its counterpart code refactoring. This makes its formalism a challenge. Existing formalization techniques [14], [15] are not declarative enough to support the detection and correction of architectural smells provided by Roock and Lippert in their book on refactoring large software projects [5] and other architecture smells provided by Stal [3] and Dietrich et al. [8].

Therefore, in order to cope with the problems of manual architectural refactorings, mitigate risk in performing these refactorings and lack of tool support, there is a critical need to formalize architectural refactorings. This

formalization aims at providing a formal support to a set of architectural refactorings identified by [3], [5], [8], so that these refactorings may be (semi-) automated with the help of tool support. These refactorings are not trivial code level refactorings, e.g. for achieving modularity in the architectures to take advantage of current modular techniques, we may need not only to refactor code but also build scripts, deployment descriptors and other related information. The research questions being addressed in this project are as follows [3]:

1) How can we systematically formalize the vague and abstract definitions of architecture level refactorings?
2) How to synchronize changes in the software architecture to implementation and vice versa?
3) What are the implications in providing declarative approach to deal with architectural refactorings?
4) How easy it would be to integrate the architectural refactoring support to IDEs?

III. Methodology

In order to tackle the issues discussed in previous section, we propose to formalize a set of architectural refactorings. We will then evaluate our approach by refactoring existing real world systems (dataset). The dataset with sufficient test coverage will be extracted from Qualitas Corpus [11] - a collection of open source software systems. Refactoring will be performed over the dataset. Results will be analyzed to evaluate the impact of refactoring and finally repeating the experiment by modifying refactoring if the results are not achieved.

A. Declarative Language for Architectural Refactoring

Architectural refactoring can be viewed as transformation of one model to another model having the same or different meta-models. In this project we need a declarative language to transform a model into another model. This language needs to be expressive enough to describe the patterns to be found in graphs and transformations in graphs. Figure 1 shows the big picture of the refactoring process. This process consists of three phases. In *simulation* phase, models will be extracted from programs and refactoring will be performed on models. In *application* phase, scripts will be generated from refactoring (transformation model) and will be applied to the actual program to refactor it. In *verification* phase, the refactored program and the target model will be matched for consistency.

The declarative language for refactoring needs to be expressive enough to gather information about a refactoring e.g. it should be able to describe patterns and transformations in graphs. It should also be able to define higher order constructs such as edge or path constraints, AST manipulation and description of graph properties. This should have the parsing facility in order to generate build scripts from transformation model, that will be used to perform refactoring on program level. Tool support is also required to automate the transformation process.

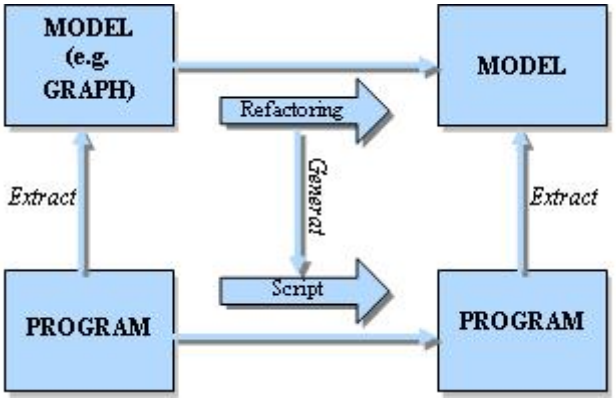

Fig. 1. Architectural Refactoring Process

We also need to define the meta model for refactoring language. We have so far short listed a list of candidates for meta modeling e.g. *XML schema* can be used to define a meta model. The XML documents will represent the model. *Eclipse Modeling Framework (EMF)* can also be used to define the meta model. We can define the refactoring contents as different shapes and the relationship between contents as edges. This model can then be used to generate CASE tools through EMF code generation facility. EMF also provides the ability to describe additional semantics of the models with OCL extensions. *Extended Backus-Naur Form (EBNF)* is a notation for expressing context free grammars for languages. We can define the grammar for our declarative language for refactoring.

B. Composite Refactorings

The idea behind composition of refactorings is to perform a series of primitive refactorings to achieve a large and complex refactoring. Refactorings have pre-conditions representing initial conditions that must be met in order to perform refactoring and post-conditions that must be met in order to complete the refactoring successfully. We propose to use the post-condition of a refactoring as a pre-condition of the next refactoring in the chain of composite refactoring. Since each primitive refactoring is behavior preserving then the composition should also be behavior preserving.

C. Fitness Functions

Fitness functions are based on a set of metrics that can be used to assess the impact of refactoring on quality of design. Improvement in fitness functions is important because the goal of refactoring is to improve the quality attributes. We propose to use the fitness functions that address the following quality attributes: modularity, maintainability, reusability, modifiability, testability. Fitness functions should be improved after the refactoring process. We propose to use some of the relevant object oriented metrics defined by Martin [16], Kemerer and Chidamber [17].

D. Invariants

An invariant is a predicate that remains the same before and after the execution of a program. In order to check the correctness of refactoring, various approaches are available. We propose to use a pragmatic approach based on testing and type-safety for checking the correctness of refactoring. This is due to the reason that we have to evaluate our project against a set of existing real world systems. In this scenario, the only available and accessible means to judge the program behavior and correctness are type checking and test cases. However this idea is not sufficient to prove the program correctness but the use of testcases is seen to be good enough by system developers. This also reflects their preference of choosing what is relevant to them. Therefore, we propose to use testcases and type checking as a verification method for refactoring process.

IV. Evaluation

This section establishes the criteria by which we intend to judge the success of our proposed work. In our experiment, we propose to use 20% of corpus programs with the highest test coverage. This is due to the reason that not all programs have test cases to verify program correctness after transformation.

Fitness functions are a set of metrics used as a measurement of the impact of refactoring on the software architecture quality. Results can be validated with the numerical measurement of these metrics before and after applying architectural refactorings such that these fitness functions will improve. The process of refactoring will be considered as successful if we are able to improve the fitness functions after applying refactoring.

We will make use of test cases available in the existing systems to ensure the behavior preservation and refactoring correctness, because test cases reflect the actual requirements and output required by software stakeholders. So if a refactoring brings abnormal behavior in the system, then test cases are not passed successfully. On the other hand when test cases are passed successfully it is likely that behavior is not changed. Therefore the property of behavior preservation can be judged by testing. Therefore all the test cases of a system need to run successfully after applying refactoring and the program compilation should be successful in order to check the type-safety.

In conclusion to all this discussion, we define the success of project when we are able to (semi-) automatically refactor the selected representation of programs, such that the behavior (Invariant) is preserved 100% and 80% of the programs are improved by improving some of the fitness functions.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[2] F. Bourqun and R. K. Keller, "High-impact refactoring based on architecture violations," in *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 149–158.

[3] M. Stal, "Refactoring of software architecture," OOPSLA 2008, 2008. [Online]. Available: http://www.oopsla.org/oopsla2008/tutorials.html

[4] "Graph Query Language 4 JUNG (GQL4JUNG)." [Online]. Available: http://code.google.com/p/gql4jung/

[5] M. Lippert and S. Roock, *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley, 2006.

[6] "IBM, BEA and JBoss adopting OSGi." [Online]. Available: http://www.infoq.com/news/2008/02/osgi_jee

[7] "Project jigsaw." [Online]. Available: http://openjdk.java.net/projects/jigsaw/

[8] J. Dietrich, C. McCartin, E. Tempero, and S. M. A. Shah, "Barriers to modularity: An empirical study to assess the potential for modularisation of java programs," in *Submitted in (ICSE) 2010, Cape Town, South Africa*, 2010.

[9] T. Mens, S. Demeyer, and D. Janssens, "Formalising behaviour preserving program transformations," in *International Conference on Graph Transformation (ICGT 2002)*. Springer-Verlag, 2002, pp. 286–301.

[10] J. Kerievsky, *Refactoring to Patterns*. Pearson Higher Education, 2004.

[11] "Qualitas research group, qualitas corpus version 20090202,," University of Auckland,, February 2009. [Online]. Available: http://www.cs.auckland.ac.nz/~ewan/corpus

[12] W. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, USA, 1992.

[13] D. Roberts, "Practical analysis for refactoring," Ph.D. dissertation, University of Illinois at Urbana-Champaign, USA, 1999.

[14] D. L. Mtayer, "Describing software architecture styles using graph grammars." *IEEE Trans. Software Eng.*, vol. 24, no. 7, pp. 521–533, 1998. [Online]. Available: http://dblp.uni-trier.de/db/journals/tse/tse24.html#Metayer98

[15] H. Fahmy and R. C. Holt, "Using graph rewriting to specify software architectural transformations," in *In Proc. of Automated Software Engineering (ASE*, 2000, pp. 187–196.

[16] R. Martin, "Object oriented design quality metrics: An analysis of dependencies," Report on object analysis and design, 1994.

[17] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," Piscataway, NJ, USA, pp. 476–493, 1994.