Quality-Driven Object-Oriented Code Restructuring *

Ladan Tahvildari
Dept. of Electrical and Computer Eng.
University of Waterloo
Waterloo, Ontario
Canada, N2L 3G1
ltahvild@swen.uwaterloo.ca

Abstract

The technique of refactoring, restructuring the source code of an object-oriented program without changing it external behavior, has been embraced by many object-oriented software developers as a way to accommodate changing requirements. It is expected that such refactoring operations or software transformations improve the maintainability of software. Unfortunately, it in unclear how specific quality factors are affected by applying such refactorings operations or software transformations. This position paper addresses the issue of developing a restructuring methodology and its associated architecture for migrating an object-oriented legacy system to a new target system. The methodology allows for specific design and quality requirements of the target migrant system to be considered during the restructuring process that is applied on iterative and incremental steps.

1 Background

An intrinsic property of software in a real-world environment is its need to evolve. As the software is enhanced, modified and adopted to new requirements, the code becomes more and more complex and drifts away from its original design. Because of this, the major part of the total software development cost is devoted to software maintenance. Better software development tools and methods 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 complexity, there is an urgent need for techniques that reduce software complexity by incrementally improving the internal software structure. The research domain that addresses this problem is referred to as *restructuring* [1] or in the case of object-oriented software development, *refactoring* [6].

Although it is possible to refactor manually, tool support is considered crucial. Today, a wide range of tools is available that automate various aspects of refactorings. Depending on the tool and the kind of support that is provided, the degree of automation can vary. Tool such as the *Refactoring Browser* [13], and *jFactor* [9] support a semi-automatic approach. Some researchers demonstrated the feasibility of fully-automated refactoring [3, 11, 14, 19]. There is also a tendency to integrate refactoring tools directly into industrial strength software development environments. Specific tools include : *Clone Doctor* [5], *IntelliJ IDEA* [8], and InjectJ [7].

The focus of all these tools in on applying a refactoring or restructuring upon request of the user. There is much less support available for detecting where and when a refactoring operation can be applied.

2 Position Statement

Quality is software domain specific. Most refactoring tools and restructuring activities only provide support for *applying* a specific refactoring, but not for finding out *where* and *why* a particular refactoring or software transformation should be applied to improve desired qualities.

This means that not much effort has been invested for systematically documenting quality attributes as a guide for the restructuring process through refactoring operations or software transformations. In this context, we need a comprehensive framework that allows for the definition and enactment of the restructuring process. Such a framework can allow for specific quality requirements of the target system to be considered during the restructuring process. This paper discusses such a proposed framework.

*This work was funded by the IBM Canada Ltd. Laboratory, Center for Advanced Studies in Toronto; also by the Ontario Graduate Scholarship (OGS) of Canada.

3 Proposed Approach

The restructuring of object-oriented systems requires a comprehensive framework to relate refactoring operations and software transformations with non-functional requirements. The focal point of the proposed research is to exploit the synergy between the areas of *software requirements analysis* [20], *software architecture* [2], and *source code analysis and transformation* [1, 10]. Effective system maintenance and evolution requires the understanding both of the inner-workings of the system (*i.e.*, its structure) and the rationale or requirements that justify its evolution. Understanding the architecture and the source code of an existing system aids in assessing the impact specific refactoring operations and software transformations have on the object-oriented system.

This means that our code-improving restructuring approach consists of : *i*) *requirements analysis phase* to identify specific quality goals, *ii*) *model analysis phase* to understand the system's design and architecture, *iii*) *source code analysis phase* to understand a system's implementation, *iv*) *remediation specification phase* to examine the particular problem and to select the optimal transformation for the system, *v*) *transformation phase* to apply transformation rules in order to restructure a system in a way that complies with specific quality criteria, and *vi*) *evaluation phase* to assess whether the transformation or refactoring has addressed the specific requirements set [18].

3.1 A Model for Quality Representation

To represent information about different software qualities, their interdependencies, and the refactorings and software transformations that may affect them, we adopt the NFR framework proposed in [4]. The NFR Framework introduces the concept of *soft-goals* whose achievement judged by the sufficiency of contributions from other (sub-)soft-goals. According to the NFR framework, software qualities are represented as soft-goals, *i.e.*, goals that can be partially achieved. A *soft-goal interdependency graph* (SIG) is used to support the systematic, goal oriented process of architectural design. The leaves of the soft-goal interdependency graph represent design decisions which fulfil or contribute positively/negatively to soft-goals above them. Given a quality constraint for a restructuring problem, one can look up the soft-goal interdependency graph for that quality, and examine how it relates to other soft-goals, and what are additional refactoring operations or software transformations that may affect the desired quality positively or negatively [18].

Each soft-goal interdependency graph (SIG) for a non-functional requirement can be considered as a directed graph (digraph) D represented as a set R of a 3-tuple elements $< N, E, L >$ where N is a set of vertices or nodes, divided into *NFR soft-goal* nodes and *transformation operationalization* nodes. The set of nodes or vertices is called the *vertex-set* of D, denoted by $N = V(D)$. There is also a special node called the *entry node*. E is a set of edges or arcs which is a list of ordered pair of the nodes. The list of arcs is called the *arc-list* of D, denoted by $E = A(D)$. If n_i and n_j are vertices, then an arc of the form $n_i n_j$ is said to be directed from n_i to n_j, or to join n_i to n_j. In this case, node n_j is said to be a *successor* of node n_i and node n_i is said to be a *parent* of node n_j. Finally, L is a labelling of $N \times E$ which assigns to each node a node of D, and to each edge an impact rule.

Let n_1 and n_2 be vertices of a SIG. If n_1 and n_2 are joined by an arc e with any of AND or OR contribution operator, then n_1 and n_2 are said to be *adjacent* with the defined rule. If the arc e is directed from n_1 to n_2, then the arc e is said to be *incident from* n_1 and *incident to* n_2. Let D to be a SIG in digraph form with n vertices or soft-goal nodes, $N = \{d_1, d_2, ..., d_n\}$. The simplest graph representation scheme uses an $n \times n$ matrix SAM (Soft-Goal Adjacency Matrix) of $\&$'s (for AND), $|$'s (for OR), and 0's.

Let n_1 be a soft-goal node and n_2 a software transformation or refactoring operation. If n_1 and n_2 are joined by an arc e with any of $MAKE$ $(++)$, $HELP$ $(+)$, $HURT$ $(-)$, or $BREAK$ $(--)$ contribution operator, then n_1 and n_2 are said to be *adjacent* with the defined rule. The arc e which is directed from n_2 to n_1 is said to be *incident from* n_2 and *incident to* n_1. Let S to be a set nodes, $S = \{s_1, s_2, ..., s_n\}$, representing soft-goals and T is a of set operationalization nodes, $T = \{t_1, t_2, ..., t_m\}$, representing transformations or refactorings. The simplest impact representation scheme uses an $m \times n$ matrix TIM (Transformation Impact Matrix) of $++$'s, $+$'s, $--$'s, $-$'s, and 0's.

3.2 A Model for Quality Assessment

So far, this research work on improving the quality of object-oriented systems through restructuring includes : i) building soft-goal interdependency graphs as a means to model the associations of qualities with design decisions and source code features, and ii) proposing a software transformation framework using refactorings based on SIGs [16]. The next step is to assign object-oriented metrics to software features and correspondingly to the transformations that affect these features in a systematic manner in order to identify the refactoring operations or software transformations that may be appropriate in improving quality as this is estimated by the selected metrics.

As it is known, an object model has several levels of representation, including *application level*, *subsystem level*, *class level*, and *function level*. While design flaws can oc-

cur at any level, our focus here is on class level deterioration. We believe that this is the most fundamental level that constitutes a system. Improving deteriorated classes should help to keep object-oriented systems operational. One way to detect design flaws at the class level is to identify violations of a "good" object-oriented software design by performing source code analysis. While there are several reasons for a class to lose quality over time, here the focus is on the classes that have high coupling and low cohesion. These characteristics often result in loss of abstraction and encapsulation. They are those highly coupled classes that often loose cohesion during the course of development [17].

Each of the design flaws and each of the quality rules for detecting these flaws are modelled as an attribute or a characteristic of a design. These characteristics are sufficiently well defined to be objectively assessed by using one or more object-oriented metrics. Metrics are particularly suitable to check, whether the object-oriented legacy system adheres to design principles or contains violations of these principles. The proposed selection of the object-oriented metrics is classified according to four major metrics categories : complexity metrics, coupling metrics, cohesion metrics, and inheritance metrics.

Consider, AC to be a set of classes in an object model extracted from an object-oriented legacy system. We need to calculate the metrics values from a predefined catalogue and apply quality heuristics rules to detect design flaws and *deteriorated classes* in the legacy system being analyzed. In this process, the first step is to apply the *key classes (KCH)* rule by using both a complexity and coupling metrics. A very high level quality goal for a software system could be maintainability, thus coupling measurements should not be high in order to ensure that changes to the system do not trigger changes throughout the system. Therefore, monitoring *DAC* values can be promising. When a significant number of classes evolves to higher *DAC* measurements, some *refactoring operations* [6] or meta-pattern transformations [16] of the system could be appropriate, to reduce coupling. Also, by applying cohesion metrics like *TCC* and coupling metrics like *DAC* and *RFC* to the object-oriented legacy system, possible violations of the principle *one class - one concept (OC2H)* rule can be found. These classes tend to have either low *TCC* values or high *DAC* and high *RFC* values. For example, classes that have very low *TCC* values, can often be split using Extract Class refactoring operation [6]. This leads to a more flexible design, since the two separate classes are easier to understand and are more reusable.

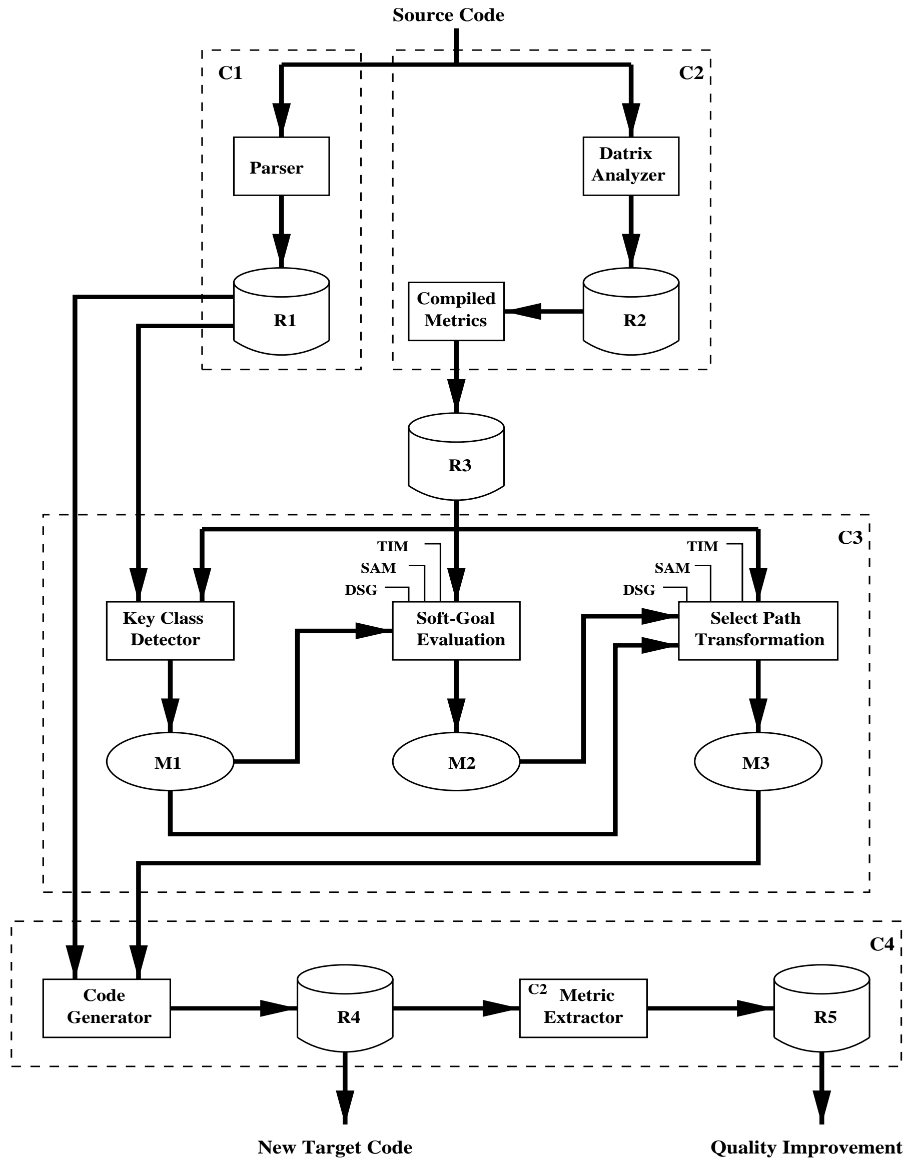
3.3 A Model for Transformation Path Selection

Now, we are particularly interested to determine a proper set of software transformations or refactoring operations among several alternative ones that can be applied on a system as means to restructure its object-oriented source code so that the new migrant system conforms with specific non-functional requirement enhancements. We have adopted a multi-objective graph search approach to identify the set of all non-dominated solution graphs in SIGs for selecting the source-code improving transformations. This algorithm can be considered as an adaptation of A^* algorithm [12]. The interested readers can refer to [15] for more information.

4 Quality-Driven OO Restructuring Process

The architectural design of the proposed QDCR approach as depicted in Figure 1 consists a number of components with simple interface and with a *pipe and filter* architectural style. Each component (filter) processes its input data in the form of a file (pipe) and stores the results in another file for the next component. Figure 1 illustrates the its components namely : i) *Pre-Process Components* (including $C1$ and $C2$ phases), ii) *Analysis Components* ($C3$), iii) *Post-Process Components* ($C4$). The proposed QDCR framework uses the incremental and iterative process as describing in the following steps :

- **Step 1 :** After parsing the source code, we measure and record the specific object-oriented metrics in order to detect classes for which quality has deteriorated. While there are several reasons for a design to lose quality over time, here the focus is on detecting the classes that have high complexity and high coupling.
- **Step 2 :** After detecting design flaws automatically, we need to re-engineer the classes using proper transformations. On the other side, the re-engineering activities need to be done based on specific non-functional requirements that are represented as a list of Desired Soft-Goals (DSG). For correcting such design flaws through software transformations or refactorings, we look at the impact of specific transformations have on specific metrics [17]. *Soft-Goal Evaluation* component builds a search graph of potential transformations based on SAM, TIM, the proposed algebraic framework for the proposed layered transformation framework, and the list of DSG. Based on this built search space, a set of all non-dominated solution graphs is identified for selecting source-code improving transformations [15].
- **Step 3 :** After applying refactorings or transformations, we need to re-apply and record the same object-oriented metrics to the restructured classes and finally compare the recorded results to evaluate design and source code improvement as well as compliance with the desired requirements.



Memory		Repository		Components	
M1:	Key classes to be transformed	R1:	Parsed code	C1:	Parsing Phase
M2:	Potentail transformation as a finite directed graph	R2:	Class/File level metrics	C2:	Metric Extractor Phase
M3:	A set of non-dominated solution graph	R3:	Indirect metrics	C3:	Model Analysis Phase
		R4:	Target code	C4:	Evaluation Phase
		R5:	Evaluated results		

Figure 1. The Architectural Design of the Quality-Driven OO Code Restructuring.

5 Conclusion and Future Work

This position paper addresses the issue of developing a restructuring methodology and its associated architecture for migrating an object-oriented legacy system to a new target system. The methodology allows for specific design and quality requirements of the target migrant system to be considered during the restructuring process that is applied on iterative and incremental steps. The focal point is to recover an object model from the source code and incrementally refine it, taking into consideration design and quality requirements for the target system.

Such a proposed QDCR framework is important for two reasons. Firstly, it attempts to address a problem that has challenged the research community for several years, namely the maintenance of object-oriented systems. Secondly, it devises a workbench in which restructuring activities do not occur in a vacuum, but can be evaluated and fine-tuned in order to address specific quality requirements for the new target system.

Several open issues have arisen from this research work. Automated component-based development is emerging as a field of study in software engineering. There are many open issues that need to be resolved before a component-based development approach can make a significant impact on mission-critical software. Methods must be developed that allow measurement and prediction of non-functional characteristics such as availability, adaptability, security, and performance. Therefore, the ability to formally model and reason about the non-functional characteristics of component-based systems is vital to any endeavour to automate the process of component development, adaptation, integration and deployment. The proposed QDCR framework can be a foundation to build such a model for component-based systems.

Refactoring or code restructuring implies that program "behavior" is preserved, but a precise definition of behavior is rarely provided or may be too inefficient to be checked in practice. Therefore, there is a need to provide a framework for restructuring with a theoretical basis in formal language theory.

Also, meta-programming techniques may be used to specify quality-related heuristics about object-oriented software (such as the detection of "bad code smells") to find out where refactorings or software transformations should be applied.

Refactoring can also be very useful at an even more implementation dependent level than programming languages, such as refactoring of byte-code, or even refactoring of executable code. Research on runtime compilers could be used in combination of quality-driven code restructuring and refactoring to adopt running applications to changing requirements without restarting them.

References

[1] R. S. Arnold. *Software Re-engineering*. IEEE Computer Society Press, 1993.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003.

[3] E. Casais. An incremental class reorganization approach. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 114–132, June 1992.

[4] L. K. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Publishing, 2000.

[5] Clone doctor (clonedr). URL = http://www.semdesigns.com/Products/.

[6] M. Fowler. *Refactoring : Improving the Design of Existing Code*. Addison-Wesley, 1999.

[7] Injectj. URL = http://injectj.fzi.de.

[8] Intellij idea 3.0. URL = http://www.intellij.com/idea.

[9] jfactor. URL = http://www.instantiations.com/jfactor/.

[10] H. W. Miller. *Re-engineering legacy software systems*. Digital Press, 1998.

[11] I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings of the ACM 11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 235–250, San Jose, California, October 1996.

[12] N. J. Nilsson. *Principles of Artificial Intelligence*. Tiago Publishing Company, 1980.

[13] D. Roberts, J. Brant, and R. Johnson. A refactoring tools for smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[14] B. Schulz. Design patterns as operators implemented with refactoring. In *Proceedings of the Workshop on Experiences in Object-Oriented Re-Engineering, European Conference on Object-Oriented Programming (ECOOP)*, Brussels, Belgium, July 1998.

[15] L. Tahvildari. *Quality-Driven Object-Oriented Re-engineering Framework*. PhD thesis, Department of E&CE, University of Waterloo, Canada, 2003.

[16] L. Tahvildari and K. Kontogiannis. A software transformation framework for quality-driven object-oriented re-engineering. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 596–605, Montreal, Canada, October 2002.

[17] L. Tahvildari and K. Kontogiannis. A metric-based approach to enhance design quality through meta-pattern transformations. In *Proceedings of the IEEE 7th European Conference on Software Maintenance and Re-engineering (CSMR)*, pages 183–192, Benevento, Italy, March 2003.

[18] L. Tahvildari, K. Kontogiannis, and J. Mylopoulos. Quality-driven software re-engineering. *Journal of Systems and Software, Special Issue on : Software Architecture - Engineering Quality Attributes*, 66(3):225–239, June 2003.

[19] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. In *Proceedings of the IEEE 14th International Conference on Automated Software Engineering (ASE)*, pages 174–181, Cocoa Beach, Florida, October 1999.

[20] R. Wieringa. *Requirements Engineering : Frameworks for Understanding*. John Wiley & Sons, 1996.