Requirements-Driven Software Re-engineering Framework

Ladan Tahvildari, Kostas Kontogiannis
Department of Elect. & Comp. Eng.
University of Waterloo
Waterloo, Ontario
N2L 3G1, Canada
{ltahvild,kostas}@swen.uwaterloo.ca

John Mylopoulos
Department of Computer Science
University of Toronto
Toronto, Ontario
M5S 3G4, Canada
jm@cs.toronto.edu

Abstract

Software re-engineering projects such as migrating code from one platform to another, or restructuring a monolithic system into a modular architecture are popular maintenance tasks. Usually, projects of this type have to conform to hard and soft quality constraints (or non-functional requirements) such as "the migrant system must run as fast as the original", or "the new system should be more maintainable than the original". This paper proposes a framework that allows for specific design and quality requirements (performance and maintainability) of the target migrant system to be considered during the re-engineering process. Quality requirements for the migrant system can be encoded using soft-goal interdependency graphs and be associated with specific software transformations that need to be carried out for achieving the target quality requirement. These transformations can be applied as a series of the iterative and incremental steps that pertain both to the design (architecture) and source code (implementation) levels. An evaluation procedure can be used at each transformation step to determine whether specific goals have been achieved.

1. Introduction

Legacy system re-engineering has emerged as a business critical task activity over the past few years. Given the amount of human effort required to manually re-engineer even a medium-sized system, most re-engineering methodologies have come to rely extensively on tools in order to reduce the human effort required. Not surprisingly, the topic of software re-engineering has been researched heavily for some time, leading to a variety of commercial toolsets for particular re-engineering tasks [7, 30], as well as research prototypes which significantly advance the state-of-the-art [9, 24].

In most cases, software re-engineering tasks have to conform to hard and soft quality constraints (or non-functional requirements) such as "the re-engineered system must run as fast as the original", or "the new system should be more easily maintainable than the original". These desired qualities (or, more precisely, desired deltas on software qualities) should play a fundamental role in defining the re-engineering process and the tools that support it. Unfortunately, there is little understanding of what this role is and how to fit it in the re-engineering process. In the research reported here, we are interested in developing a framework that uses quality requirements to define and guide the re-engineering process.

Section 2 of the paper addresses the problem more precisely. Section 3 of the paper adopts the Non-Functional Requirements (NFR) framework to represent software qualities and their interdependencies. This section also reports on our efforts to catalogue performance and maintainability qualities as well as relevant transformations that have been proposed in the literature, using the NFR framework. Section 4 discusses the proposed quality-driven software re-engineering process. Section 5 describes a set of experiments for evaluating the effectiveness of different transformations on *performance* and *maintainability* at both the source code and architectural levels, while Section 6 presents and discusses the results of these experiments by trying them out on two different software systems. Finally, Section 8 summarizes the contributions of this research and outlines directions for further research.

2. Problem Definition

The scenario that we want to assume is as follows: an existing legacy system is being re-engineered in order to conform with a new requirement (*i.e.*, performance enhancement). After studying the code and the desired requirement, it is concluded that the existing structure of the program makes the desired extension difficult to achieve,

1095-1350/01 /$10.00 © 2001 IEEE

and that the application of some design patterns or source code transformations would help to achieve the desired property. In this context, we aim to provide an automatic support for the developer as much as possible. In this approach the developer can decide what design pattern or transformation to apply towards achieving a specific non-functional requirement for the new system.

To denote the problem more precisely, we assume that the re-engineering process consists of a series of transformations $t_1, t_2, ..., t_n$ on the abstract syntax tree $AST(S)$ [1] of a software system S. We also assume that for each quality of interest, say Q, there is a metric MQ which measures how well a software system (or system fragment) fares with respect to the quality. Examples of software properties for the migrant system include "Software is written in Java", while examples of qualities include "time and space characteristics", "maintainability", "portability", "customizability", and the like. A quality-based re-engineering problem is defined as follows:

Given a software system S, relevant quality metrics $MQ_1, MQ_2, ..., MQ_n$, a desired software property P, and a set of constraints $C_1, C_2, ..., C_t$ on the software qualities $Q_1, Q_2, ..., Q_n$, find a sequence of transformations $t_1, t_2, ..., t_n$ such that the new re-engineered system $S' = t_n(t_{n-1}(...(t_1(AST(S)...)$ is such that $P(S')$ and the constraints $C_1(S'), C_2(S'), ..., C_t(S')$ hold.

To tackle the problem posed above in a more systematic fashion, we need to resolve several issues. Firstly, we need a catalogue of heuristic transformations and refactoring operations [10, 27] which are relevant to particular qualities, *e.g.*, elimination of *GOTOs* is relevant to maintainability, while procedure in-lining is relevant to performance. Secondly, we need to know how such transformations would affect other software qualities. For instance, we would like to know how *GOTO* elimination would affect performance or portability. Finally, we need quantitative data on the impact of a particular transformation on a particular quality. If we applied the transformation once, would this improve the quality metric by 2% or 10%? This also raises another issue: if we relied on such quantitative data, what would we need to measure?

This paper reports on initial results addressing some of these questions. The results include a catalog of the transformations that affect *performance* and *maintainability* quality requirements.

3. A Framework for Soft-goals

To represent information about different software qualities, their interdependencies, and the software transformations that affect them, we adopt the Non-Functional Requirement (NFR) framework discussed in detail in [5]. In the NFR framework, quality requirements are treated as potentially conflicting or synergistic goals to be achieved, and are used to guide and rationalize the various design decisions taken during system development. Because quality requirements are soft and subjective by nature, they are often achieved not in an absolute sense, but to a sufficient or satisfactory extent. Accordingly, the NFR framework introduces the concept of *soft-goals* whose achievement judged by the sufficiency of contributions from other (sub-) soft-goals. A *soft-goal interdependency graph* is used to support the systematic, goal-oriented process of architectural design. Soft-goals can be related to other soft-goals in terms of relations such as AND, OR, $+$, $++$, or $-$, $--$. The meaning of these relations is as follows:

- $AND(G, G_1, G_2, ..., G_n)$ – soft-goal G is fulfilled when all of $G_1, G_2, ..., G_n$ are fulfilled and there is no negative evidence against it.
- $OR(G, G_1, G_2, ..., G_n)$ – soft-goal G is fulfilled when one of $G1, G_2, ..., G_n$ is fulfilled and there is no negative evidence against it.
- $+(G_1, G_2)$ – soft-goal G_1 contributes positively to the fulfillment of soft-goal G_2.
- $-(G_1, G_2)$ – soft-goal G_1 contributes negatively to the fulfillment of soft-goal G_2.

According to the framework, software qualities are represented as soft-goals, *i.e.*, goals that can be partially achieved. The leafs of the soft-goal interdependency graph represent transformations which fulfill or contribute positively/negatively to soft-goals above them. Given a quality constraint for a re-engineering problem, one can look up the soft-goal interdependency graph for that quality, and examine how it relates to other soft-goals, also what are the transformations that may affect it positively or negatively. Transformations are also represented as soft-goals (which are fulfilled when they are included in the re-engineering process). When a transformation makes a contribution towards one or more parent soft-goals, it is related to the latter in terms of a link labeled $+$, $++$, or $-$, $--$. An example of soft-goal interdependency graph is that the transformation *"dead code elimination"* contributes very positively ($++$) to the soft-goals high *control flow consistency* and *high data consistency*. It also contributes positively ($+$) to the performance soft-goals *low main memory utilization*, and *low secondary storage utilization*.

3.1. Maintainability Soft-goals

Maintainability is defined as the quality of a software system that relates with the ease of adding new functionality named *perfective maintenance*, porting the system from one platform to another named *adaptive maintenance*, or

fixing errors named *corrective maintenance* [14]. Maintainability can be evaluated by the system revision history and source code measurements. Specifically, it can be measured as a quantification of the time necessary to make maintenance changes to the product, or as a percentage of code affected [16, 26] when a code fix or modification is applied.

As described in 3, the leafs of the soft-goal interdependency graph represent transformations which fulfill or contribute positively/negatively to soft-goals above them. In this context, Figure 1 shows portions of a soft-goal interdependency graph. This graph attempts to represent and organize a comprehensive set of software attributes that relate to software i maintainability. The graph was compiled after a thorough review of the literature [12, 28]. In the figure, AND relations are represented with a single arc, and OR relations with a double arc. It is important to note that in this work we only describe maintainability soft-goals that are relevant to the source code. Other maintainability related soft-goals, *e.g.*,, management related ones are not included here.

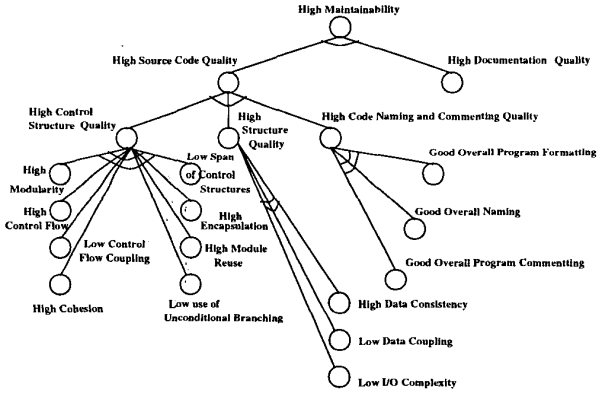

Figure 1. Maintainability Soft-Goal Graph.

3.2. Performance Soft-goals

Similarly to maintainability, performance-related requirements and their interdependencies are represented in terms of a soft-goal interdependency graph. In Figure 2, the high performance soft-goal is AND decomposed into time performance, and space performance [25]. The time performance soft-goal is OR decomposed into low response time and high throughput. The rest of the performance related decompositions are illustrated in Figure 2 which only shows a small portion of the graphs that have been compiled through a review of the literature [13, 25].

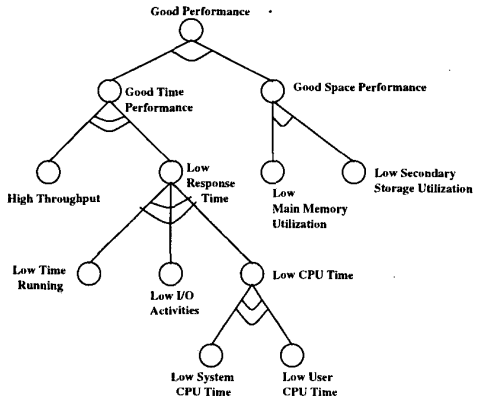

Figure 2. Performance Soft-Goal Graph.

4. The Quality-Driven Re-engineering Process

This research develops a software re-engineering model that is driven by specific non-functional requirements. The major theme to the proposed approach is to exploit the synergy between *requirements analysis* [36], *software architecture* [13], and *reverse engineering* [4]. Understanding the architecture of an existing system aids in predicting the impact evolutionary changes have on specific quality characteristics of the system [31]. Requirements analysis techniques, in turn, suggest what concepts are most useful in understanding how an existing system works and how it should evolve. This research proposal builds on this synergy by developing a re-engineering model that can be applied at two different levels of abstraction namely, *architectural* and *source code* levels.

It means that our re-engineering approach should consist of: i) *requirements analysis* to identify specific re-engineering goals, ii) *model analysis* to understand the system's design and architecture, iii) *source code analysis* to understand a system's implementation, iv) *remediation specification* to examine the particular problem and to select the optimal transformation for the system, v) *transformation* to apply transformation rules in order to re-engineer a system in a way that complies with specific quality criteria, and vi) evaluation process to assess whether the transformation has addressed the specific requirements set [32].

The re-engineering process includes the following steps as shown in Figure 3. First, the source code is represented as an Abstract Syntax Tree [1]. The tree is decorated using a linker, with annotations that provide linkage, scope, and type information. In a nutshell, once software artifacts have been understood, classified and stored during the reverse engineering phase, their behavior can be readily available to the system during the forward engineering phase. Then, the forward engineering phase aims to produce a new version of legacy system that operates on the target architecture and

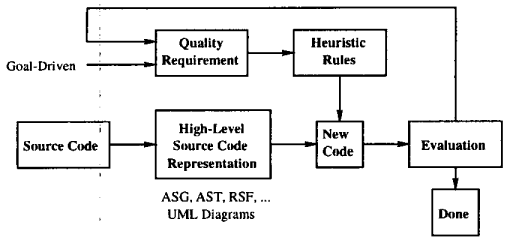

Figure 3. Quality-Based Re-engineering.

meets specific non-functional requirements (*i.e.*, maintainability or performance enhancements). Finally, we use an iterative procedure to obtain the new migrant source code by selecting and applying a transformation which leads to performance or maintainability enhancements. The transformation is selected from the soft-goal interdependency graphs. The resulting migrant system is then evaluated and the step is repeated until quality requirements are met [32]. It means that the proposed process is iterative and incremental in nature.

5. Transformations Rules

This section describes the selected transformation rules at both the source code and the architectural levels for our goal-driven re-engineering process.

5.1. Transformations at Source Code Level

The transformation rules that have been chosen in this study at source code level for enhancing maintainability are as follows:

- *Maximize Cohesion:* The number of input-output flows for a given module (*i.e.*, a class and its corresponding public methods) is minimized by reorganizing source code and by splitting methods and functions.
- *Minimize Coupling:* The number of data dependencies between methods (*i.e.*, due to global variables) is minimized by introducing parameter passing and eliminating global data flow dependencies between modules.
- *Goto Elimination:* Goto statements in the source code are replaced by iterative structures involving do, for and while loops. This type of transformations produces structured code at the expense of introducing more statements in the re-engineered code having a direct negative effect on performance.
- *Global Data Type Elimination:* Global data types that have been only used in specific modules are redeclared within the scope/module that are used. Moreover, data structures that are not used at all are eliminated.

For maintainability measurements, we focus on source code features both for the re-engineered and the original source code. In particular, we adopted the LOC (Lines Of Code), the Halstead suite of metrics [17], and McCabe's cyclomatic complexity metric [22] whose objective is to determine the number of paths through a program that must be tested to ensure complete coverage and to measure the difficulty of understanding a program. These metrics not only have been found to correlate highly with source code complexity, but also provide the basic information for computing maintainability indices. In order to determine the maintainability of the migrant code with respect to the maintainability of the original code, we calculate the Software Maintainability Index (SMI) computed in three different ways. The first (Method A) is based on Halstead's effort metrics [26]. The second and third ways (Method B [21] and Method C [6]) are both based on Halstead's effort, V(G) (McCabe's cyclomatic complexity), LOC (Lines Of Code), and CMT (number of comment lines per module).

The transformation rules that have been chosen in the study at source code level for enhancing performance are as follows:

- *Function Inlining:* Here function or method calls are replaced with inlined source code. Note that even if some compilers attempt to perform automatically function inlining, this works only for complete programs (with a main routine) and can not be applied selectively.
- *Bit Shifts vs. Integer Division and Multiplication:* Here division and multiplication operations in the source code, are replaced by bit shifts.
- *Loop Invariants and Sub-expression Elimination:* This transformation simplifies expressions inside an iterative statement (for, do, and while loops) with respect to initialization of variables that can occur outside the scope of the iterative statement. We experimented with sub-expression elimination in cases where the compiler could not perform this well-known optimization (*e.g.*, on conflicts between updates and uses of variables that occur in common sub-expressions) [1].
- *Access optimization:* The objective of this performance optimization activity is to fit all the global scalar variables of each system in a global variable pool. Then, each of the global scalar variables get accessed via pointer and an offset, instead of via constant address. This way, more expensive load and store sequences are avoided and code size is reduced [2].

- *Elimination of dead code:* Eliminates routines or libraries that are not invoked or used anywhere.

The performance analysis is based on these two measurements: i) *time* command which returns total execution time used by the program itself, as well as the system on behalf of the program, ii) *Dhrystone Benchmark* which generates a number that can serve as a comparison measure on how optimizations applied by the compiler relate to the system's performance [34]. In this context, the highest the number, the more an optimization may affect positively system performance.

5.2. Transformations at Architectural Level

In addition to the transformations at the source code level described above, we have also considered a number of design patterns transformations at the architectural level. Moreover, we have evaluated their impact on performance and maintainability on the migrant code. In particular, we have introduced six different design patterns, from three different categories of [11] as follows:

- *Creational Patterns* are concerned with the class instantiation process. They become important as systems evolve to depend more on object composition than class inheritance. As that happens, emphasis shifts away from hard-coding a fixed set of behaviors toward defining a smaller set of fundamental behaviors that can be composed into any number of more complex ones. We selected two patterns from this category namely, *Factory Method*, and *Abstract Factory* as they are two common ways to parameterize a system by the classes of objects it creates.

- *Structural Patterns* are concerned with how classes and objects are composed to form larger structures. The *Composite* pattern which describes how to build a class hierarchy, and the *Facade* pattern which allows the re-architecting a software system into subsystems and helping minimize the communication and data flow dependencies between subsystems were selected.

- *Behavioral Patterns* are concerned with the algorithms and assignment of the responsibilities between objects. The *Iterator* pattern which allows to access an aggregate object's contents without exposing its internal representation, and the *Visitor* pattern which allows to perform operations on objects that compose containers were chosen.

6. Experiments

In this section, we apply the proposed quality-driven re-engineering framework on two medium-size systems. First, the the two case studies will be described and then the collected results are presented and discussed at both the source code and architectural levels.

Our experiments were carried on a SUN Ultra 10 (440MHZ, 256M memory, 512 swap disk) in a single user mode. We use Rigi [24, 23] for extracting facts from the source code in order to provide a high-level view of systems. We also use Together/C++ UML Editor [33] to provide an interface to the source code generated by the Object-Orientation Migration Tool [29]. For collecting software metrics, we use Datrix Tool [8].

6.1. Case Studies

We have applied the quality-driven re-engineering framework described in Section 4 on the following two medium-size software systems:

1. The WELTAB Election Tabulation System [35] that was created in the late 1970s to support the collection, reporting, and certification of election results by city and county clerks' offices in USA. It was originally written in an extended version of Fortran on IBM and Amdahl mainframes under the University of Michigan's MTS operating system. At various times through the 1980s, it was run on Comshare's Commander II time-sharing service on a Xerox Sigma machine, and on IBM 4331 and IPL (IBM 4341 clone) machines under VM/CMS. Each move caused inevitable modifications in the evolution of the code. Later, the system was converted to C and run on PCs under MS-DOS. The latest version of the system is composed of 4.25 KLOC and 35 batch files. Specifically, there are 26 header files, 39 source code files, and the rest are data files for a total of 190 files.

2. The GNU AVL Libraries is the second system we have migrated to an object-oriented platform using the soft-goal NFR interdependency graph. This system is a public domain library written in C for sparse arrays, AVL, Splay Trees, and Binary Search Trees [15]. The library also includes code for implementing single and double linked lists. The original system was organized around C structs and an elaborate collection of macros for implementing tree traversals, and simulating polymorphic behavior for inserting, deleting and tree re-balancing operations. The system is composed of 4 KLOC of C code, distributed in 6 source files and 3 library files.

As discussed in 4, we adopt an incremental and iterative re-engineering process that is driven by the soft-goal interdependency graphs presented in Section 5. During each step, we select a transformation, apply it to the code, and

Table 1. Extracted Simple Object Model After Migration from C to C++.

System	Transformation Rules	Perf. Time (%diff)	Perf. Dhrystone (%diff)	SMI Method A [26] (%diff)	SMI Method B [21] (%diff)	SMI Method C [6] (%diff)
AVL	Simple Object Model Extraction	−84.14	7.75	1.46	26.69	13.39
WELTAB	Simple Object Model Extraction	−24.14	2.49	4.43	12.84	9.35

Table 2. Impact of Source Code Level Transformations on Performance.

System	Transformation Rules	Perf. Time (%diff)	Perf. Dhrystone (%diff)	SMI Method A [26] (%diff)	SMI Method B [21] (%diff)	SMI Method C [6] (%diff)
AVL/WELTAB	Function In-lining	21.30	4.24	−4.16	−10.89	−12.39
AVL/WELTAB	Integer Division	14.81	3.37	−0.20	−0.80	−0.23
AVL	Loop Invariants	14.68	2.26	−0.69	−2.81	−1.97
WELTAB	Access Optimization	8.20	1.39	−0.10	−0.76	−0.19
WELTAB	Dead Code Elimination	6.21	3.14	6.23	38.30	20.49

then obtain measurements related to the maintainability and performance of the new system.

6.2. Discussions for Source Code Level Transformations

Using the transformations described in Section 5.1, we have collected experimental results in order to evaluate the impact of particular transformations. Table 1, summarizes experimental results obtained by comparing the performance of the original system and the re-engineered system before the application of any performance transformations. The results indicate that the new $C++$ versions of the subject systems are on average 54% slower that the original system implemented in C. The observed performance degradation is largely due to the frequent invocation of object constructors, the elimination of the macros in the original code, and the consequent introduction of class hierarchies and run-time resolves polymorphic methods. Similarly, the Dhrystone number indicates that the migration to an object-oriented platform allows the compiler to perform better optimizations that may relate to higher performance (positive increase in the Dhrystone number). Along the same trend are the maintainability measurements that indicate that the new $C++$ systems are more maintainable than the original C systems.

The experimental results of Table 2 summarize the impact of different transformations on system performance. The results indicate that the highest positive performance impact is due to function inlining, integer division optimization and loop invariants. Specifically, function inlining reduces the overhead associated with traps to the kernel, and simplifies the process stacks for passing parameters and returning and results for methods or function invocations. Similarly, the integer division transformation uses hardware-based optimizations to enhance the performance of division expressions. As far as the impact of transformations on the optimizations the compiler can apply, the Dhrystone number indicates that in all cases the transformations allow for the compiler to generate binary code that has the potential for higher performance (positive increase in the Dhrystone number).

On the other hand, performance related transformations are shown to degrade maintainability, as this is measured by the three different indices presented in Section 5.1. In particular, function inlining is found to be the most "expensive" transformation with respect to maintainability due to the side effect of introducing to a given code block a significant amount of new statements, operators, and instructions. Moreover, it creates unstructured code and therefore reduces its understandability. Similarly, loop invariants, integer division, and access optimization have no significant effect on maintainability. Finally, dead code elimination increases both performance and maintainability.

This section summarizes the experimental results obtained by applying the maintainability transformations presented in Section 5.1. In particular, Table 3 summarizes our findings on the application of five transformations, namely dead code elimination, global data type elimination, minimizing coupling, maximizing cohesion, and GOTO elimination. The experimental results indicate that dead code elimination is an excellent transformation for maintainabil-

Table 3. Impact of Source Code Level Transformations on Maintainability.

System	Transformation Rules	Perf. Time (%diff)	Perf. Dhrystone (%diff)	SMI Method A [26] (%diff)	SMI Method B [21] (%diff)	SMI Method C [6] (%diff)
WELTAB	Dead Code Elimination	6.21	4.14	6.23	38.3	20.49
WELTAB	Global Data Type Elimination	−3.02	7.61	7.27	4.59	0.65
AVL	Minimizing Coupling	−3.83	14.64	4.37	4.63	4.78
AVL	Maximizing Cohesion	−2.99	12.11	1.07	1.89	2.20
WELTAB	GOTO Elimination	−33.91	4.88	−0.59	−6.82	−3.54

Table 4. Impact of Architectural Level Transformations on Maintainability and Performance.

System	Pattern Name	Perf. Time (%diff)	Perf. Dhrystone (%diff)	SMI Method A [26] (%diff)	SMI Method B [21] (%diff)	SMI Method C [6] (%diff)
AVL/WELTAB	Abstract Factory	−6.48	7.82	−4.78	−7.22	−2.09
AVL/WELTAB	Factory Method	7.04	7.50	−1.94	−3.70	−1.19
AVL/WELTAB	Composite	5.35	0.55	11.53	5.24	2.69
AVL/WELTAB	Facade	11.14	6.12	9.82	8.14	3.91
AVL/WELTAB	Iterator	−4.44	4.78	8.96	12.72	0.72
AVL/WELTAB	Visitor	4.08	4.87	7.44	5.81	0.88

ity because it minimizes the overall size of the code and simplifies control and data flow of a module. Similarly, global data type elimination allows for the minimization of redundant data flow dependencies, due to inclusion of libraries. Coupling and cohesion are two other properties that are confirmed to affect positively maintainability without significantly affecting negatively system performance. On the contrary, GOTO elimination affects negatively the maintainability indices. This occurs because GOTO elimination introduces more code (iterative statements).

6.3. Discussions for Architectural Level Transformations

Table 4 shows the results of applying transformations that relate to design patterns. Also, Figure 4 and Figure 5 depict the target system WELTAB before and after applying those patterns. We have used the Together/C++ UML Editor [33] in order to show the various relations between the obtained classes.

- **Creational Patterns :** There are two common ways to parameterize a system in terms of the classes of objects it creates. One way is to subclass the class that invokes the appropriate constructors. This corresponds to using the factory method pattern. Our experimental results indicate an average of 2.5% decrease of maintainability by applying the creational design patterns as shown in Table 4. Meanwhile, the application of these patterns is shown to provide better performance at the range of almost 7.04% on average. The factory method pattern can make a design more customizable. Often, industrial software system designs start by using the factory method and evolve towards other creational patterns as the designer discovers the points where more flexibility is required.

  The other way to parameterize a system relies more on object composition. Specifically, the pattern allows for the definition of a class that supports constructors that can be parameterized. This is a key aspect of abstract factory patterns which involve creating a new "factory object" whose responsibility is to create product objects by invoking their corresponding constructors. Comparisons between Figure 4 and Figure 5 clarify the existence of this "factory object". Figure 4 depicts the object model obtained without considering any design patterns (simple object model extraction), while Figure 5 illustrates the same system after applying the selected design patterns.

  For example, consider the "REPORT" class in Figure 4 which is used to generate multiple reports. Different reports have different appearances and headers for printing. To be portable across reports, an application should not be hard-coded for a particular report. We can solve this problem by defining an abstract "Report-

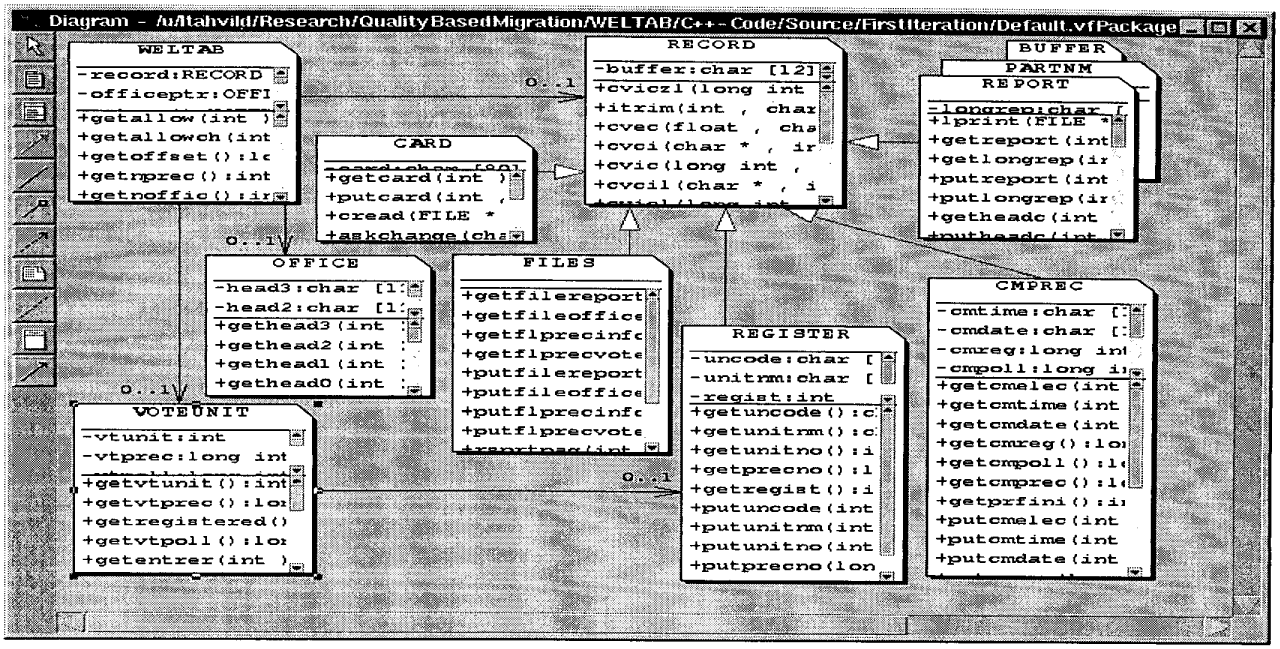

Figure 4. Object Model of WELTABIII.

Gen" class that declares an interface for creating each kind of reports as shown in Figure 5. This class acts as an Abstract Factory Pattern. Our results indicate that the Abstract Factory Pattern reduces the performance by almost 7%, possibly due to the dynamic nature of selecting an invoking the proper constructors. A design that uses the Abstract Factory Pattern is even more customizable than those that use the Factory Method Pattern. However, the incorporation of this pattern has been shown to decrease maintainability by an average of almost 4.5%. This is a typical case of conflicts in the NFR soft-goal interdependency graph (customizability vs. performance and maintainability).

- **Structural Patterns:** The Composite Pattern describes how to build a class hierarchy that is made up of two kinds of objects: primitives and composites. It means that the key to the Composite Pattern is an abstract class that represents both primitives and their containers. This pattern is found to improve maintainability because it allows for component sharing. Experimental results indicate maintainability improvement at the levels of 6% on average. Similarly, this pattern allows for performance increase as well, because it allows for explicit superclass references and simplifies component interfaces. Our experimental results confirm this result for an average increase of 5%.

- **Behavioral Patterns:** For this part, we have used the Iterator Pattern to access an aggregate object's contents without exposing its internal representation. The pattern also supports multiple traversals of aggregate objects. Providing a uniform interface for traversing different aggregate structures is another reason to use this pattern that supports polymorphic iteration.

  Our experimental results indicate an average increase of maintainability at the level of 7%. However, the pattern reduces performance by almost 4% because of more than one traversal can be pending on an aggregate object. Moreover, we used the Visitor Pattern whenever we wanted to perform operations on objects that compose containers. This pattern makes it easy to add operations that can be applied in an iterative way and depend on the components of complex objects. A new operation over an object structure can be added by simply adding a new visitor class. This pattern helps making the system more maintainable. Our results show an average improvement of 4.5% for maintainability indices. The Visitor Pattern helps in applying operations to objects that don't have a common parent class. This results in a reduction of traversal time, and makes it a good transformation to use for performance enhancements.

7. Related Work

Over the last 30 years, a number of researchers and practitioners have examine how systems achieve software qual-

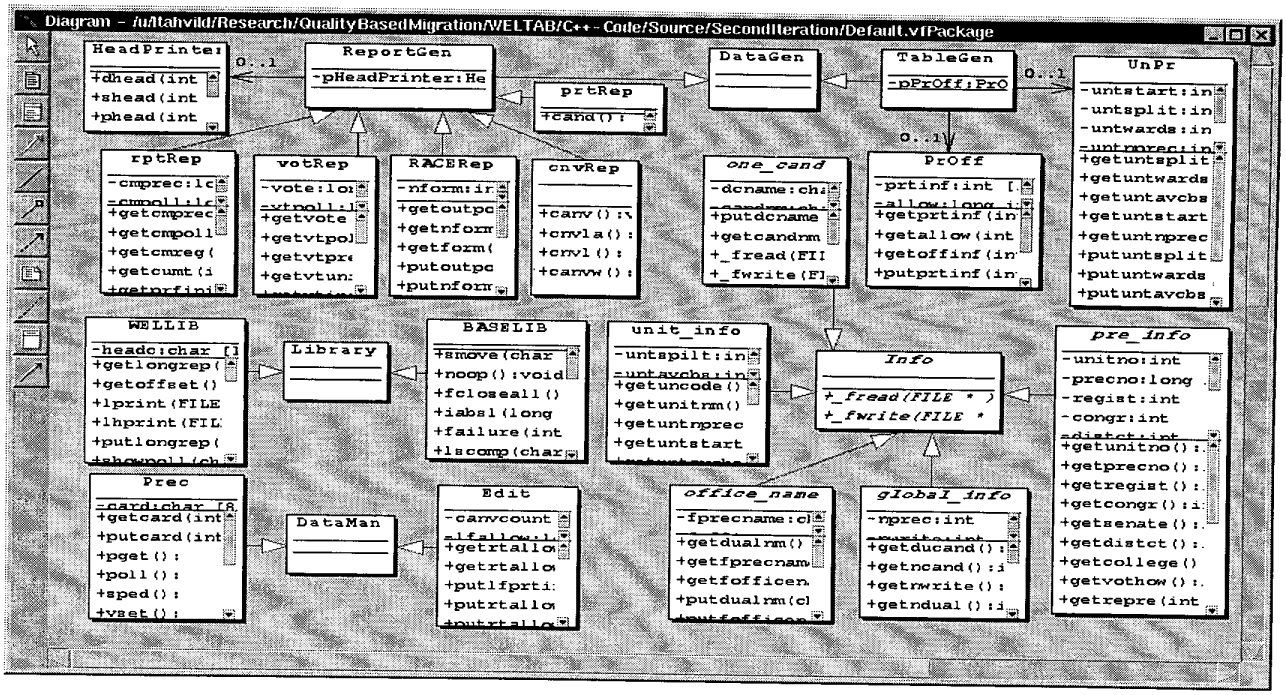

Figure 5. Object Model of WELTABIII with Design Patterns.

ity attributes. Boehm and International Organization for Standardization (ISO) introduced taxonomies of quality attributes [3, 18]. The Software Engineering Institutes (SEI) work in Attribute-Based Architecture Style (ABAS) [19] was the first attempt to document the relationship between architecture and quality attributes. However, no one has systematically and completely documented the quality attributes as a guidance for the software re-engineering process.

The problem of coping with qualities during re-engineering has been experimentally tackled by developing a number of tools that met particular quality requirements. In [20], a tool-set has been developed that assists on the migration of PL/IX legacy code to $C++$ while maintaining comparable time performance. [29] describes a re-engineering tool that supports the transformation of C code to $C++$ code that is consistent with object-oriented programming principles. In both cases, the approach was experimental. First, a tool has been built to perform the re-engineering task, then a trial-and-error strategy was used to select a particular set of transformations which ensured that the re-engineered code satisfied given quality constraints.

8. Conclusion

We have proposed a framework for software re-engineering which is requirements-driven in the sense that it uses desirable qualities for the re-engineered code to define and guide the re-engineering process. This framework is noteworthy for two main reasons. First, it attempts to address a problem that challenges the research community for several years, namely the maintenance of object-oriented mission critical systems. Second, it aims to devise a workbench in which re-engineering 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 such as, enhancements in maintainability, and performance.

The contributions of the paper include an initial compilation of factors and transformations which affect two particular software qualities, performance and maintainability. As well, we have presented a framework for experimentally evaluating the effect of different transformations on different qualities, and have collected initial results using two medium-size software systems.

Further research is required to refine the framework and make it more readily usable. In particular, we are working on extensions of the framework that allow the estimation of the impact of a transformation on the qualities of a particular software system. We are also investigating a new approach to software metrics which is founded on soft-goal interdependency graphs.

Acknowledgments

This work was funded by the IBM Canada Ltd. Laboratory, Center for Advanced Studies in Toronto, and also by

the Natural Sciences and Engineering Council (NSERC) of Canada. The authors also thank William Andreopoulos of the University of Toronto for his help in the early version of this paper.

References

[1] A. V. Aho, R. Sethi, and U. Jeffrey. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[2] D. F. Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, Computer Science Division, University of California, Berkeley, 1997.

[3] B. Boehm et al. *Characteristics of Software Quality*. Elsevier North-Holland Publishing Company, Inc., 1978.

[4] E. J. Chikofsky and J. H. CrossII. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, pages 13–17, January 1990.

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

[6] D. Coleman, B. Lowther, and P. Oman. The Application of Software Maintainability Models in Industrial Software Systems. *The Journal of Systems and Software*, 29:3–16, 1995.

[7] J. R. Cordy and I. H. Carmichael. The TXL Programming Language Syntax and Semantics Version 7. Technical Report 93-355, Department of Computing and Information Sciences, Queen's University, Kingston, Canada, June 1993.

[8] Datrix Metric Reference Manual, Version 4.1. Bell Canada, 2000. Also available at http://www.iro.umontreal.ca/labs/gelo/datrix.

[9] P. Finnigan et al. The Software Bookshelf. *IBM Systems Journal*, 36(4):564–593, November 1997.

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

[11] E. Gamma, R. Helm, R. Jahnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[12] D. Garlan, G. E. Kaiser, and D. Notkin. Using Tool Abstraction to Compose System. *IEEE Computer*, 25:30–38, June 1992.

[13] D. Garlan and M. Shaw. *An Introduction to Software Architecture*. World Scientific Publishing Co., 1993.

[14] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 1991.

[15] GNU AVL Libraries. Also available at http://www.interads.co.uk/~crh/ubiqx.

[16] J. R. Hagemeister. *A Metric Approach to Assessing the Maintainability of Software*. PhD thesis, Department of Computer Science, University of Idaho, 1992.

[17] M. Halstead. *Elements of Software Science*. Elsevier North-Holland Inc., 1977.

[18] International Organization for Standardization (ISO). Information Technology, Software Product Evaluation, Quality Characteristics and Guidelines for Their Use, ISO/IEC 9126, 1996.

[19] M. Klein, L. Bass, and R. Kazman. Attribute-Based Architecture Styles. Technical Report CMU/SEI-99-TR-022 ADA371802, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1999.

[20] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Muller, and J. Mylopoulos. Code Migration Through Transformations: An Experience Report. *Proceedings of IBM CASCON'98 Conference*, pages 1–13, 1998.

[21] B. Lowther. The Application of Software Maintainability Metric Models to Industrial Software Systems. Master's thesis, Department of Computer Science, University of Idaho, 1993.

[22] T. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[23] H. Muller. Rigi as a Reverse Engineering Tool. Technical Report DCS-160-IR, University of Victoria, Victoria, BC, Canada, 1991.

[24] H. Muller, M. Orgun, S. Tilley, and J. Uhl. A Reverse Engineering Approach to Subsystem Identification. *Software Maintenance and Practice*, 5:181–204, 1993.

[25] B. A. Nixon. Dealing with Performance Requirements During the Development of Information Systems. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 42–49, January 1993.

[26] P. Oman and J. R. Hagemeister. Constructing and Testing of Polynomials Predicting Software Maintainability. *The Journal of Systems and Software*, 24:251–266, 1994.

[27] W. Opdyke. *Refactoring Object-Oriented Framework*. PhD thesis, University of Illinois, 1992.

[28] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM (CACM)*, 15:1053–1058, 1972.

[29] P. Patil. Migration of Procedural Systems to Object Oriented Architectures. Master's thesis, Department of Electrical and Computer Engineering, University of Waterloo, 1999.

[30] Software Refinery, Reasoning Systems. Also available at http://www.reasoning.com.

[31] L. Tahvildari, R. Gregory, and K. Kontogiannis. An Approach for Measuring Software Evolution Using Source Code Features. In *Proceedings of the IEEE Asia-Pacific Software Engineering (APSEC)*, pages 10–17, December 1999.

[32] L. Tahvildari and K. Kontogiannis. A Workbench for Quality Based Software Re-engineering to Object Oriented Platforms. *Proceedings of ACM International Conference in Object Oriented Programming, Systems, Languages, and Applications (OOPSLA) - Doctoral Symposium*, pages 157–158, October 2000.

[33] Together/C++ UML Editor. Also available at http://www.togethersoft.com/.

[34] R. P. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *Communications of the ACM (CACM)*, 27(10):1053–1058, 1984.

[35] WELTAB Election Tabulation System. Also available at http://pathbridge.net/reproject/cfp2.htm.

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