ELSEVIER

The Journal of Systems and Software 66 (2003) 225–239

The Journal of Systems and Software

www.elsevier.com/locate/jss

Quality-driven software re-engineering ☆

Ladan Tahvildari [a,*], Kostas Kontogiannis [a,*], John Mylopoulos [b]

[a] Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ont., Canada N2L 3G1
[b] University of Toronto, Canada

Received 22 January 2002; received in revised form 9 April 2002; accepted 15 May 2002

Abstract

Software re-engineering consists of a set of activities intended to restructure a legacy system to a new target system that conforms with hard and soft quality constraints (or non-functional requirements, NFR). This paper presents a framework that allows specific NFR such as performance and maintainability to guide the re-engineering process. Such requirements for the migrant system are modeled using soft-goal interdependency graphs and are associated with specific software transformations. Finally, an evaluation procedure at each transformation step determines whether specific qualities for the new migrant system can be achieved.

Keywords: Software re-engineering; Soft-goal interdependency graphs; Software architecture; Software quality; Non-functional requirements; Software metrics; Design patterns

1. Introduction

Over the past few years, legacy system re-engineering has emerged as a business critical activity. Software technology is evolving by leaps and bounds, and in most cases, legacy software systems need to operate on new computing platforms, be enhanced with new functionality, or be adapted to meet new user requirements. 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 computer aided software engineering tools in order to reduce human effort on performing these re-engineering tasks. Not surprisingly, the topic of software re-engineering has been researched heavily for some time, leading both to a variety of commercial tool-sets for particular re-engineering tasks (Cordy and Carmichael, 1993; Software Refinery, Reasoning Systems, 1984; Sneed, 1992), and to research prototypes (Baxter, 1990; Finnigan et al., 1997; Kazman et al., 2000; Kontogiannis et al., 1998; Műller et al., 1993).

Software re-engineering aims to achieve hard and soft quality requirements and objectives 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". For example, the Open Group identifies in (The Open Group Architecture Framework Version 7, 2001) a number of software system qualities related to evolution, which range from increasing user productivity, to improving portability and scalability, improving vendor independence, enhancing security, manageability, and more. 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 it fits in the re-engineering process. In this paper, we are interested in developing a software re-engineering model that is driven by specific non-functional requirements (NFR) and addresses issues related to the evolution of the system requirements and software architecture (Garlan

☆ This work was funded by the IBM Canada Ltd. Laboratory—Center for Advanced Studies (Toronto), by Ontario Graduate Scholarship (OGS) in Science and Technology of Canada, by the Natural Sciences and Engineering Research Council (NSERC) of Canada, and by the Institute of Robotics and Intelligent Systems (IRIS), a network of centers of excellence funded by the Government of Canada. This paper is a revised, expanded version of a paper presented at the 8th International IEEE Working Conference on Reverse Engineering (WCRE), October 2001.

* Corresponding authors. Tel.: +1-519-885-1211; fax: +1-519-746-3077.

E-mail addresses: ltahvild@swen.uwaterloo.ca (L. Tahvildari), kostas@swen.uwaterloo.ca (K. Kontogiannis).

0164-1212/03/$ - see front matter © 2002 Published by Elsevier Science Inc.
doi:10.1016/S0164-1212(02)00082-1

and Shaw, 1993). Understanding the architecture of an existing system assists on predicting the impact evolutionary changes may have on specific quality characteristics of the system (Tahvildari et al., 1999). This research bridges the gap between NFR Framework, software architecture evolution, and software transformations.

The remainder of this paper is organized as follows. Section 2 defines the problem of quality-driven re-engineering. Section 3 discusses the proposed quality-driven software re-engineering process. Section 4 presents the NFR Framework to model software qualities and their interdependencies. Section 5 reports on our efforts to catalogue performance and maintainability qualities as well as relevant architectural transformations using the NFR Framework. Section 6 presents and discusses experimental results obtained by applying the method on two different software systems. Section 7 reviews related work. Finally, Section 8 summarizes the contributions of this work and outlines directions for further research.

2. Problem definition

We assume the following scenario: an existing legacy system is being re-engineered in order to conform with a quality 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, and that the application of some design patterns, or source code transformations would help to achieve the desired property. In this context, the aim is to provide support for the developer to decide what design patterns or transformations to apply towards achieving the specific quality requirement for the new system (e.g., performance enhancement).

To denote the problem more precisely, we assume that the re-engineering process consists of a series of transformations $t_1, t_2, \ldots, t_n$ on the abstract syntax tree AST(S) (Aho et al., 1988) 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 specific quality. Examples of software properties for the migrant system include "The target application should run on NT platforms", while examples of qualities include "time and space performance", "maintainability", "portability", "customizability", and the like. A quality-based re-engineering problem can be defined as follows:

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

To address the problem presented above in a more systematic fashion, we need to resolve several research issues, namely: (i) the composition a list of software transformations which relate to particular software qualities, (ii) the investigation of the mutual impact these transformations have on software qualities, (iii) the design of a method to quantitatively assess the impact of a particular transformation on a particular quality in terms of metrics or quantitative software indices.

This paper reports on the design and development of such a re-engineering framework that aims on addressing these issues. The results include a catalog of the transformations that relate to two specific qualities of the migrant system namely, *performance* and *maintainability*, and a quantitative framework to assess the impact these transformations have on the specific qualities.

3. A quality-driven re-engineering framework

In this section, we present an evolutionary view of the software re-engineering life-cycle. This view is illustrated in Fig. 1 and is composed of six phases (Tahvildari and Kontogiannis, 2000).

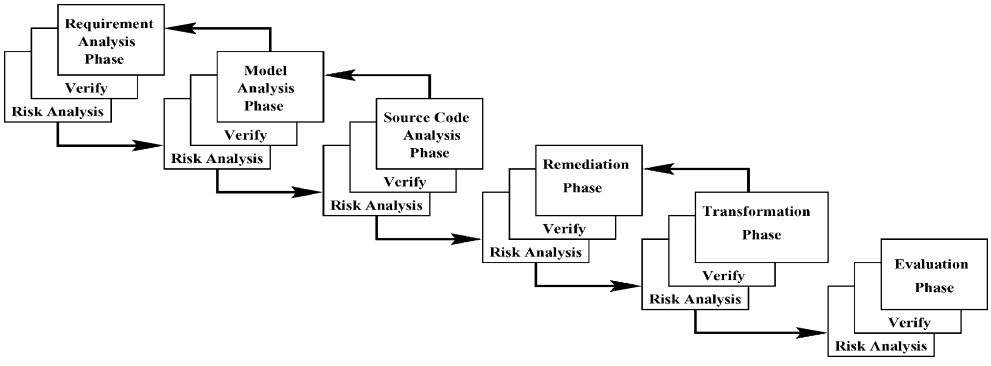

Fig. 1. The re-engineering life-cycle.

Requirements analysis phase: refers to the identification of concrete re-engineering goals for a given re-engineering project. The specification of the criteria should be specified and illustrated in the new re-engineered system (for example, faster performance). Violations that need to be repaired are also identified in this phase.

Model analysis phase: refers to documenting and understanding the architecture and the functionality of the legacy system being re-engineered. In order to understand and to transform a legacy system, it is necessary to capture its design, its architecture and the relationships between different elements of its implementation. As a consequence, a preliminary model is required in order to document the system and the rationale behind its design. This requires reverse engineering the legacy system in order to extract design information from its code.

Source code analysis phase: refers to the identification of the parts of the code that are responsible for violations of requirements originally specified in the system's analysis phase. This task encompasses the design of methods and tools to inspect, measure, rank, and visualize software structures. Detecting error prone code that deviates from its initial requirement specifications requires a way to measure where and by how much these requirements are violated. Problem detection can be based on a static analysis of the legacy system (i.e., analyzing its source code or its design structure), but it can also rely on a dynamic usage analysis of the system (i.e., an investigation of how programs behave at run-time).

Remediation phase: refers to the selection of a target software structure that aims to repair a design or a source code defect with respect to a target quality requirement. Because legacy applications have been evolved in such a way that classes, objects, and methods may heavily depend on each other, a detected problem may have to be decomposed into simpler subproblems.

Transformation phase: consists of physically transforming software structures according to the remediation strategies selected previously. This requires methods and tools to manipulate and edit software systems, re-organize and re-compile them automatically, debug and check their consistency, and manage different versions of the software system being re-engineered (Tahvildari and Singh, 1999).

Evaluation phase: refers to the process of assessing the new system as well as, establishing and integrating the revised system throughout the corporate operating environment. This might involve the need for training and possibly the need for adopting a new improved business process model.

Such a re-engineering life-cycle yields a re-engineering process as illustrated in Fig. 2. First, the source code is represented as an Abstract Syntax Tree (Aho et al., 1988). The tree is further decorated with annotations that provide linkage, scope, and type information. Once

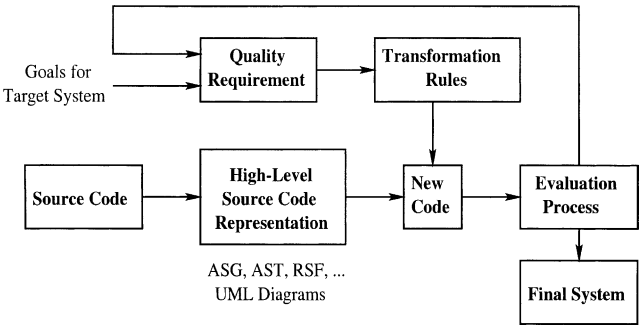

Fig. 2. The block diagram of the quality-based re-engineering process.

software artifacts have been understood, classified and stored during the reverse engineering phase (Chikofsky and CrossII, 1990), 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 a legacy system that operates on the target architecture and aims to address specific NFR (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 the specific quality requirements are met (Tahvildari and Kontogiannis, 2000, 2002; Tahvildari et al., 2001).

4. Soft-goal interdependency graphs

To represent information about different software qualities, their interdependencies, and the software transformations that may affect them, we adopt the NFR Framework proposed in (Chung et al., 2000). In the NFR Framework, quality requirements are treated as potentially conflicting or synergistic goals that need be achieved, and are used to model and rationalize the various design decisions to be taken during system/software development. Accordingly, the NFR Framework introduces the concept of *soft-goals* whose achievement is judged by the sufficiency of contributions from other (sub-) soft-goals. In this context, a *soft-goal interdependency graph* is used to support the systematic modeling of the design rationale.

For example, suppose a system developer has to design and produce source code that complies with an initial set of quality requirements, such as "the system should be modifiable" and "the system should have real-time performance". In this process-oriented approach, the developer explicitly represents each of these as a soft-goal to be achieved during the design and development process. Each soft-goal (e.g., Modifiability

[system]) is associated with a *type* (Modifiability) and a *topic* (System), along with other information such as importance, satisficing status and time of creation.

As these high-level requirements may denote different concepts to different people, the developer needs to first clarify their meanings. This is done through an iterative process of soft-goal refinement which may involve domain experts. Consequently, the developer may refine Modifiability [System] into three offspring soft-goals: Modifiability [Algorithm], Modifiability [Data Representation], and Modifiability [Function]. This refinement is based on topic, since it is the topic (System) that is being refined, while the soft-goal type (Modifiability) is unchanged. This step may be justified by referring to the work by Garlan and Shaw (1993) who consider changes in processing algorithm and changes in data representation, and to Garlan and Kaiser (Garlan et al., 1992) who extend the consideration with enhancement to system function. Similarly, the developer refines Performance [System], this time based on its type, into Space Performance [System] and Time Performance [System] (Nixon, 1993) as shown in Fig. 3 where a small "arc" between edges denotes an *AND* contribution, meaning that in order to satisfy the parent soft-goal, all of its offsprings need to be *satisficed*.

At this point, let us assume that the developer is interested in a new design that can contribute positively to the soft-goal Modifiability [Data Representation], and considers the use of an "Abstract Data Type" style as discussed by Parnas (Parnas, 1972) and Garlan (Garlan and Shaw, 1993). It means that the components can communicate with each other by means of explicit invocation of procedures as defined by component interfaces. The developer may discover sooner or later that the positive contribution of the "Abstract Data Type" towards modifiable data representation has been made at the expense of another soft-goal, namely the time performance soft-goal. Fig. 4 illustrates an example of the positive contribution made by the abstract data type solution towards Modifiability by means of a "+" and the negative contribution towards Performance by means of a "−" *contribution link*.

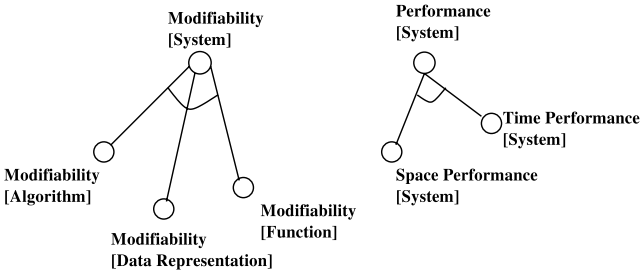

Fig. 3. A soft-goal interdependency graph.

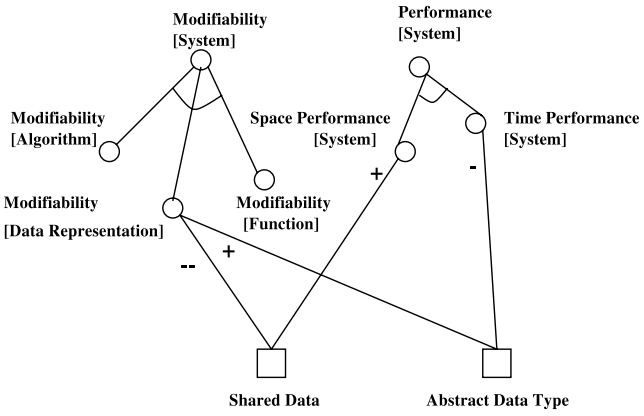

Fig. 4. Contributions of the shared data and abstract data type in soft-goal graph.

The developer may also want to consider other alternatives in order to better satisfice the stated soft-goals.

The developer may discover that a "Shared Data" style typically would not degrade system response time, at least when compared to the Abstract Data Type, and more importantly perhaps it is quite favorable with respect to space requirements. This discovery draws on work by Parnas (1972) and by Garlan (Garlan and Shaw, 1993) where the basic components (modules) communicate with each other by means of shared storage. Not unlike the Abstract Data Type, the Shared Data architecture has very negative "– –" (Chung et al., 2000) impact on modifiability of data representation. Fig. 4 illustrates both NFR steps along with the various contributions that each alternative makes towards the refined soft-goals.

The NFR Framework is one significant step towards making the relationships between quality requirements and design decisions explicit. The framework uses NFR in order to support architectural design and to model the impact of design alternatives. 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, and what are additional transformations that may affect the desired quality positively or negatively. Transformations are also represented as soft-goals which are fulfilled when they are included in the re-engineering process.

5. Architectural level transformations

For this research work, we are particularly interested to investigate design patterns and their relationships as a means to restructure an object-oriented legacy system so that the new system conforms with specific design patterns and meets specific NFR criteria. For achieving this

goal, we need to develop a list of specific design patterns and refactoring operations (Fowler, 1999) that can be used to enhance specific software qualities during re-engineering namely, *performance* and *maintainability* enhancements for the new migrant system. Performance is a vital quality factor in real-time, transaction-based, or interactive systems. Since without good performance, such systems would be practically unusable. Similarly, maintainability enhancement is another important requirement for a re-engineered system and can be partially achieved by the use of design patterns.

Fig. 5 provides an overview on how the design pattern approach is related to the proposed quality-driven re-engineering process. As Fig. 5 depicts, the components or source code features that can be directly extracted from the object-oriented legacy system, provide important information for guiding the developers to detect and apply candidate transformations that yield design patterns that may improve the quality of the target system. In a nutshell, we can relate the re-engineering model presented in Section 3 schematically as in Fig. 5. Specifically, (i) *requirements analysis* identifies specific re-engineering goals, (ii) *model analysis* aims for the understanding of the system's design and architecture, (iii) *source code analysis* aims for the understanding of a system's implementation through *Extracted Components and Features*, (iv) *remediation specification* examines the particular problem through *Refactoring Operations* and selects the optimal transformation for the system, (v) *transformation* applies transformation rules in order to re-engineer a system in a way that complies with specific quality criteria, and (vi) *evaluation process* assesses whether the transformation has addressed the specific requirements set that *Metric Analysis* can do this job.

In (Gamma et al., 1995), a catalogue of design patterns is presented. The catalogue not only lists a description of patterns but also presents how these patterns are related. Furthermore, the catalogue presents a classification of all design patterns according to two criteria: *jurisdiction* (class, object, compound) (Gamma et al., 1993) and *characterization* (creational, structural, behavioral) (Gamma et al., 1995). However, these relationships in (Gamma et al., 1995) are described informally and each relationship appears to be different in its formalization from the other ones. We propose a classification scheme (Fig. 6) of the standard design patterns (Gamma et al., 1995) in a way that we believe it can assist software maintainers to better assess the impact of these design patterns when applied to object-oriented software restructuring (Tahvildari and Kontogiannis, 2002). This scheme is based on three *primary* relationships between patterns such as: (i) a pattern *uses* another pattern, (ii) a pattern *refines* another pattern, (iii) a pattern *conflicts* with another pattern. The proposed classification also describes three *secondary* relationships between patterns such as: (i) a pattern is *similar* to another one, (ii) two patterns *combine* to solve a single problem, (iii) a pattern *requires* the solution of another pattern. We also show how these secondary relationships can be expressed in terms of the primary relationships (Tahvildari and Kontogiannis, 2002). These classifications motivate us to update the catalogue and organize the design patterns into two layers representing different abstraction levels as depicted in Fig. 6. Based on such a classification and layered catalogue, we will consider a number of design patterns transformations at the architectural level that will be discussed in the form of criteria selection in Section 6.2. Moreover, we will evaluate their impact on performance and maintainability on the migrant code.

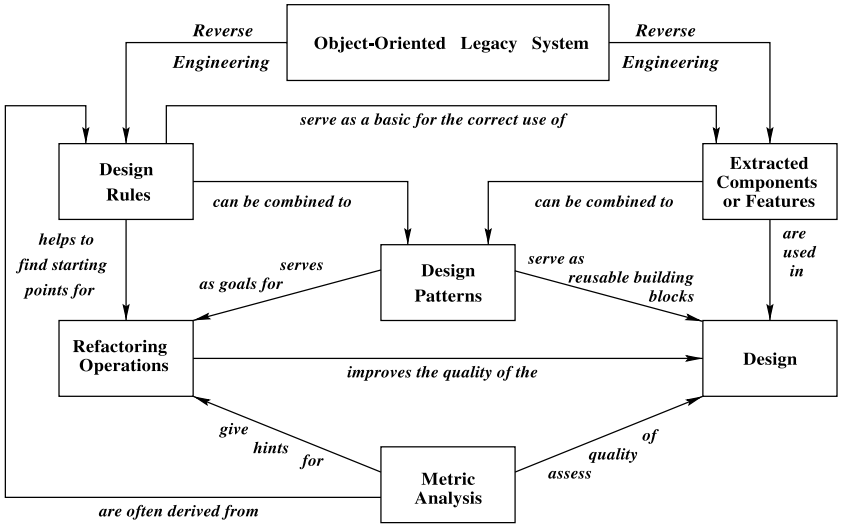

Fig. 5. Role of design patterns in software re-engineering.

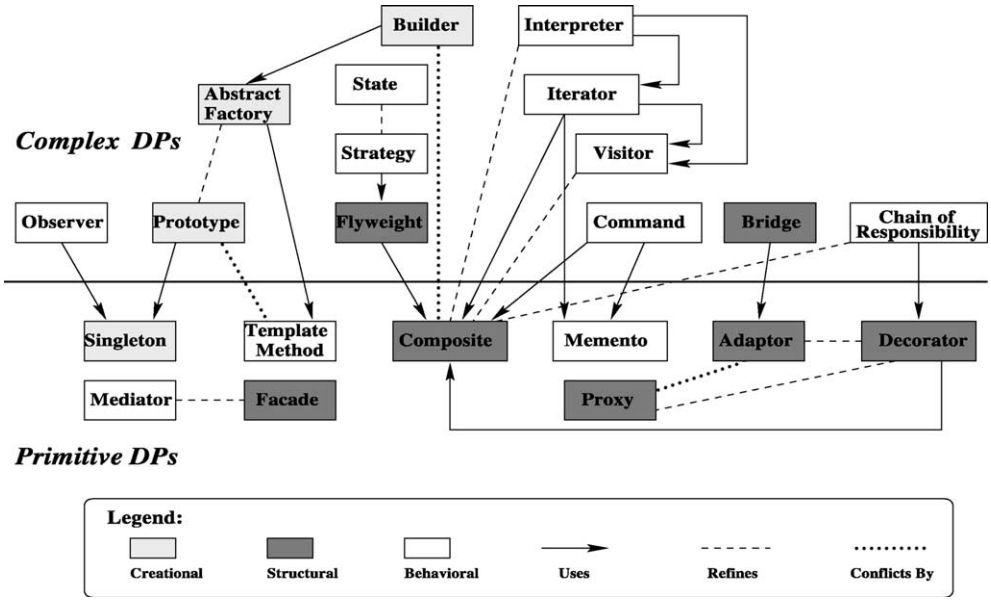

Fig. 6. Arrangement of design pattern catalogue in layers.

5.1. Maintainability soft-goals

Maintainability is defined as the quality of a software system that relates with the ease of adding new functionality (i.e. *perfective maintenance*), porting the system from one platform to another (i.e. *adaptive maintenance*), or fixing errors (i.e. *corrective maintenance*). Maintainability can be evaluated by the system revision history and source code measurements. Maintainability as a re-engineering requirement is quite broad and abstract. Various research teams have determined numerous attributes and characteristics of software systems that relate to maintainability. To effectively model how software attributes and characteristics relate to maintainability, we must provide a comprehensive and formal classification framework. Such a framework can be defined in terms of a semantic net of NFR that we call "soft-goals".

As described in Section 4, the leafs of the soft-goal interdependency graph represent transformations which fulfill or contribute positively/negatively to soft-goals above them. In this context, Fig. 7 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 maintainability and was compiled after a thorough review of the literature (Garlan et al., 1992; Parnas, 1972). In Fig. 7, *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 soft-goals relevant to the source code and the architecture of the target system. It is also possible to identify maintainability-related soft-goals that do not depend directly on source code and system architecture properties. However, identifying such soft-goals would require knowledge about specific environmental factors (such as management and process modeling issues) and are outside the scope of the work presented here.

In this context, we have classified the maintainability NFR soft-goal graph into two major areas namely, attributes that relate to source code quality (Oman and Hagemeister, 1994), and attributes that relate to the documentation quality. We argue that both source code and documentation quality soft-goals must be satisfied for a system to have high maintainability. This is referred to as an AND contribution of the offspring soft-goals towards their parent soft-goal, and is shown by grouping the interdependency lines with an arc. The source code quality soft-goal can be further decomposed into three sub-soft-goals namely, high control structure quality, high information structure quality, and high code typography, naming and commenting quality (Oman and Hagemeister, 1994). This decomposition is shown in Fig. 7, and also an AND contribution, i.e. all three sub-soft-goals must be satisfied to achieve the high source code quality soft-goal. The rest of the decompositions are also illustrated in Fig. 7.

5.2. Performance soft-goals

Similar to maintainability, performance-related requirements and their interdependencies are represented in terms of a soft-goal interdependency graph. In Fig. 8, the high performance soft-goal is *AND* decomposed into time performance, and space performance (Nixon, 1993). The time performance soft-goal is *OR* decomposed into low response time and high throughput. The rest of the performance-related decompositions are il-

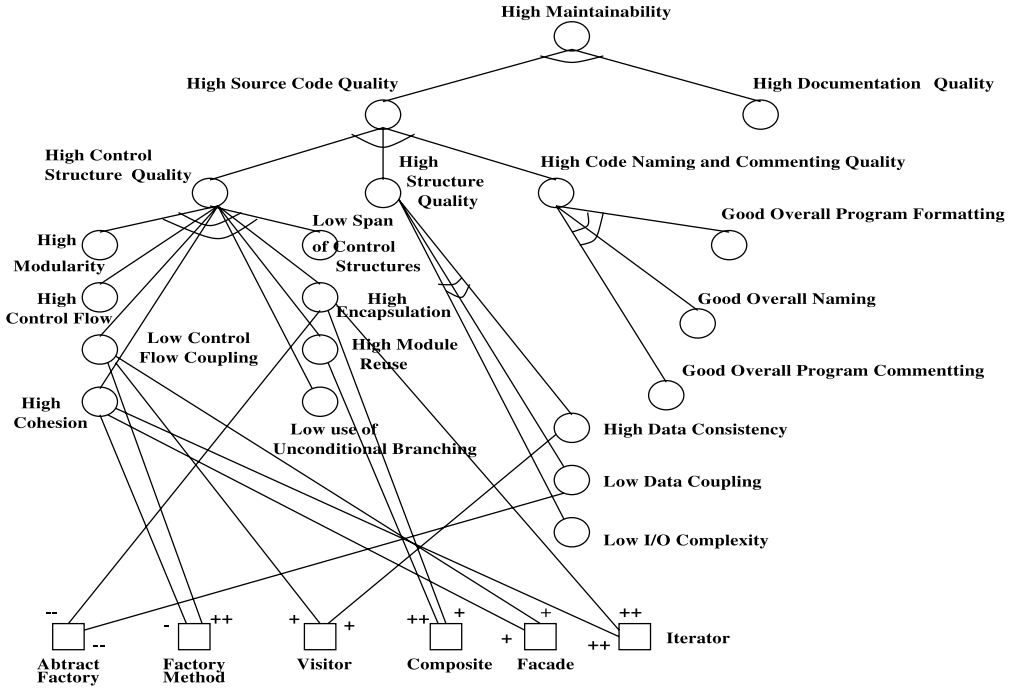

Fig. 7. Maintainability soft-goal decomposition.

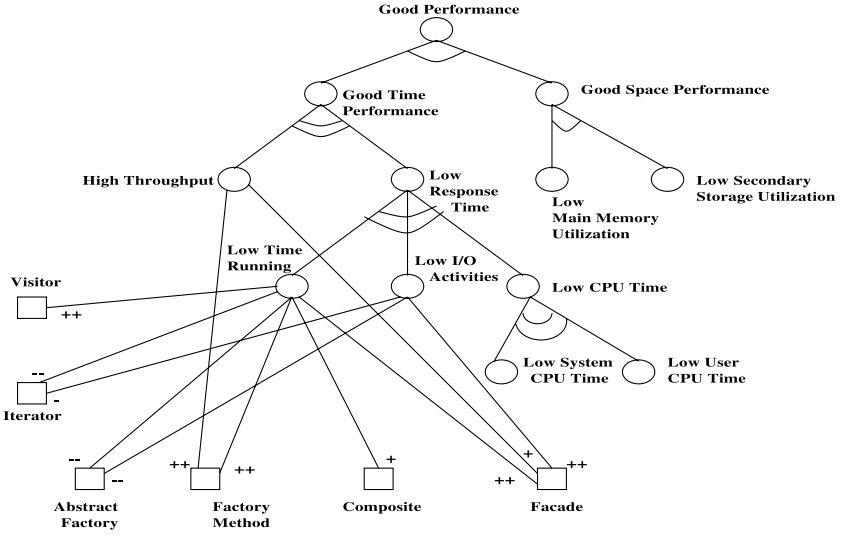

Fig. 8. Performance soft-goal decomposition.

lustrated in Fig. 8 which only shows a small portion of the graphs that have been compiled through a review of the literature (Garlan and Shaw, 1993; Nixon, 1993).

5.3. Measuring the impact of transformations on software qualities

For maintainability measurements, we focus on source code features that relate both to the re-engineered and the original source code. In particular, we adopt the lines of code (LOC), the Halstead suite of metrics (Halstead, 1977), and the McCabe's cyclomatic complexity metric (McCabe, 1976) 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 have been found to correlate highly with the difficulty to maintain a system and provide the basic information for computing maintainability indices (Coleman et al., 1995; Lowther, 1993; Oman and Hagemeister, 1994). 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 as follows:

(1) The first one, named SMI A, is a single metric model based on Halstead's efforts (Halstead, 1977), and has been proposed in (Oman and Hagemeister, 1994)

$$\text{SMI Method A} = 125 - 10\log(\text{ave} - E)$$

(2) The second one, named SMI B, is a four-metric model based on Halstead's efforts (Halstead, 1977), McCabe's $V(G)$ (McCabe, 1976), LOC, and CMT (number of CoMmenT lines per module), and has been proposed in (Lowther, 1993)

$$\begin{aligned}\text{SMI Method B} = {} & 171 - 5.44\ln(\text{ave} - E) - 0.23\,\text{avg} \\ & - V(G') - 16.2\ln(\text{avg} - \text{LOC}) \\ & + 50\sin(\text{sqrt}(2.46\,(\text{avg} \\ & - \text{CMT}/\text{avg} - \text{LOC})))\end{aligned}$$

(3) The last one, named SMI C, is a four-metric model based on Halstead's effort (Halstead, 1977), McCabe's $V(G)$ (McCabe, 1976), LOC, and CMT (number of CoMmenT lines per module), and has been proposed in (Coleman et al., 1995).

$$\begin{aligned}\text{SMI Method C} = {} & 171 - 3.42\ln(\text{ave} - E) - 0.23\,\text{avg} \\ & - V(G') - 16.2\ln(\text{avg} - \text{LOC}) \\ & + 0.99\,\text{avg} - \text{CMT}\end{aligned}$$

Similarly, the performance analysis is based on the following two measurements:

(1) *time:* which returns total execution time used by the program itself, as well as the system on behalf of the program,
(2) *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 (Weicker, 1984). In this context, the highest the number, the more an optimization may affect positively system performance.

6. Empirical evaluation

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

Our experiments were carried on a SUN Ultra 10 (440MHZ, 256M memory, 512 swap disk) in a single user mode. We use Rigi (Műller et al., 1993) for extracting facts from the source code in order to provide a high-level view of the systems. We also use the Together/C++ UML Editor (1999) to provide an interface to the source code generated by the Object-Orientation Migration Tool (Patil, 1999). The main reason of using Together is that it supports UML editing and the changes in the object model are tracked back in the source code. Finally, for collecting software metrics, we use the Datrix Tool (Datrix Metric Reference Manual, Version 4.1, 2000).

6.1. Case studies

We have applied the quality-driven re-engineering framework described in Section 3 on the following two medium-size software systems. As discussed in Section 3, we adopt an incremental and iterative re-engineering process that is driven by the soft-goal interdependency graphs presented in Section 4. During each step, we select a transformation, we apply it to the code, and then obtain measurements related to its impact towards increasing the maintainability and performance for the new system.

6.1.1. WELTAB Election Tabulation System

This system was built in the late 1970s to support the collection, reporting, and certification of election results by city and county clerks' offices in the State of Michigan (WELTAB Election Tabulation System, 1980). 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 CommanderII 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, WELTAB was converted to C and run on PCs under MS/DOS (non-GUI, pre-Windows). Modifications were often made in such a way that some sections were rewritten entirely, while most still show signs of their Fortran origin. The result is a system of C programs and command/data files. WELTAB III that has 4.5 KLOC source code and 189 KLOC samples which include 35 batch files, 26 header files, 39 source code files, and the rest are data files which provides 190 files in total. The Object-Orientation Migration Tool (Patil, 1999) has been applied to WELTABIII in order to migrate the original C source code to new object-oriented C++ code. The object model for this system is depicted in Fig. 9.

6.1.2. The GNU AVL library

The second system is a public domain library written in C for sparse arrays, AVL, Splay Trees, and Binary Search Trees (GNU AVL Libraries, 1999). The library also includes code for implementing single and double linked lists. The original system was organized around C

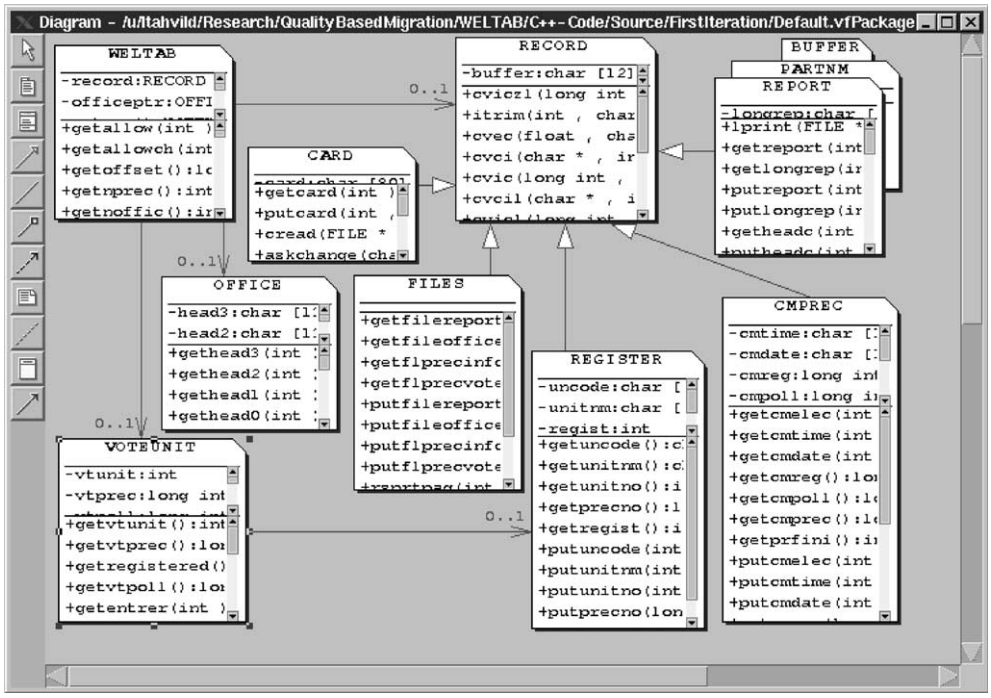

Fig. 9. Object model of WELTABIII.

structs and an elaborate collection of macros for implementing tree traversals, and simulating polymorphic behavior for inserting, deleting and tree re-balancing operations. The library consists of a set of core modules that implement basic constructs. These include lists and binary trees. Other, slightly more complex constructs are built on top of the basic ones. These includes sparse arrays and data caches. The system is composed of 8.4 KLOC of C code, distributed in six source files and three library files. The GNU AVL has been migrated in a previous project to C++ (Patil, 1999) and its object model is depicted in Fig. 10.

6.2. Selection criteria for architectural transformations

First for the restructuring transformations, we have considered a "Primitive Structural" design pattern namely, the *Composite* design pattern which is the most popular one as it is *used by* four other design patterns and is *refined by* three other design patterns as shown in Fig. 6. The *Composite* pattern describes how to build a class hierarchy that is made up of different kinds of objects. In this category, another design pattern for re-architecting a software system into subsystems and helping minimize the communication and data flow dependencies between subsystems is the *Facade* design pattern. This pattern shields clients from the subsystem components, thereby reducing the number of objects that clients deal with and making the subsystems easier to use and maintain.

Second, we have considered "Complex" design patterns that can be built on top of the *Composite* design pattern. As far as the *Behavioral patterns* are concerned, we consider two of them namely, *Iterator* and *Visitor*. The *Iterator* pattern allows 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. On the other hand, the *Visitor* pattern helps making the migrant system more maintainable because a new operation over an object structure can be modified simply only by adding a new visitor class.

Third, we have considered *Creational patterns* as they are concerned with the class instantiation process. They become more 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. This set of fundamental behaviors can be composed to produce more complex ones. There are two common ways to parameterize a system by 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* patterns. 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

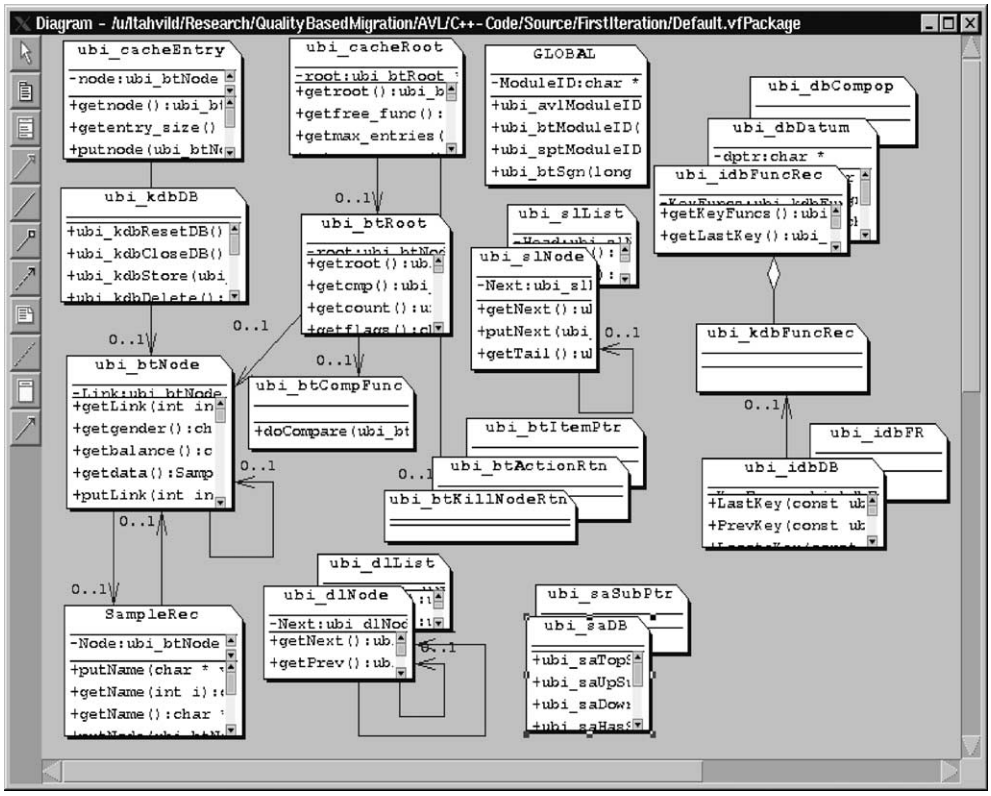

Fig. 10. Object model of AVL GNU library.

of *Abstract Factory* patterns which involve creating a new "factory object" whose responsibility is to generate product objects by invoking their corresponding constructors.

6.3. Discussions on architectural transformation experiments

Using the transformations described in Section 6.2, 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 are obtained by earlier transformations that allowed for the migration of the original WELTAB system to C++ (Patil, 1999). Using these earlier transformations, the new C++ versions of the subject systems were 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.

Having presented these initial benchmarks, we summarize the experimental results obtained by applying some of the design pattern transformations presented in this paper and discussed in detail in Section 5. The results and the impact on performance and maintainability by applying selected design patterns are illustrated in Table 2. Moreover, Figs. 9–11 depict our target systems before and after applying those design pattern transformations. Detail discussions and interpretations of the results are included in the following subsections.

Table 1
Extracted simple object model after migration from C to C++

System	Perf. time (% diff)	Perf. Dhrystone (% diff)	SMI Method A (% diff)	SMI Method B (% diff)	SMI Method C (% diff)
AVL	−84.14	7.75	1.46	26.69	13.39
WELTAB	−24.14	2.49	4.43	12.84	9.35

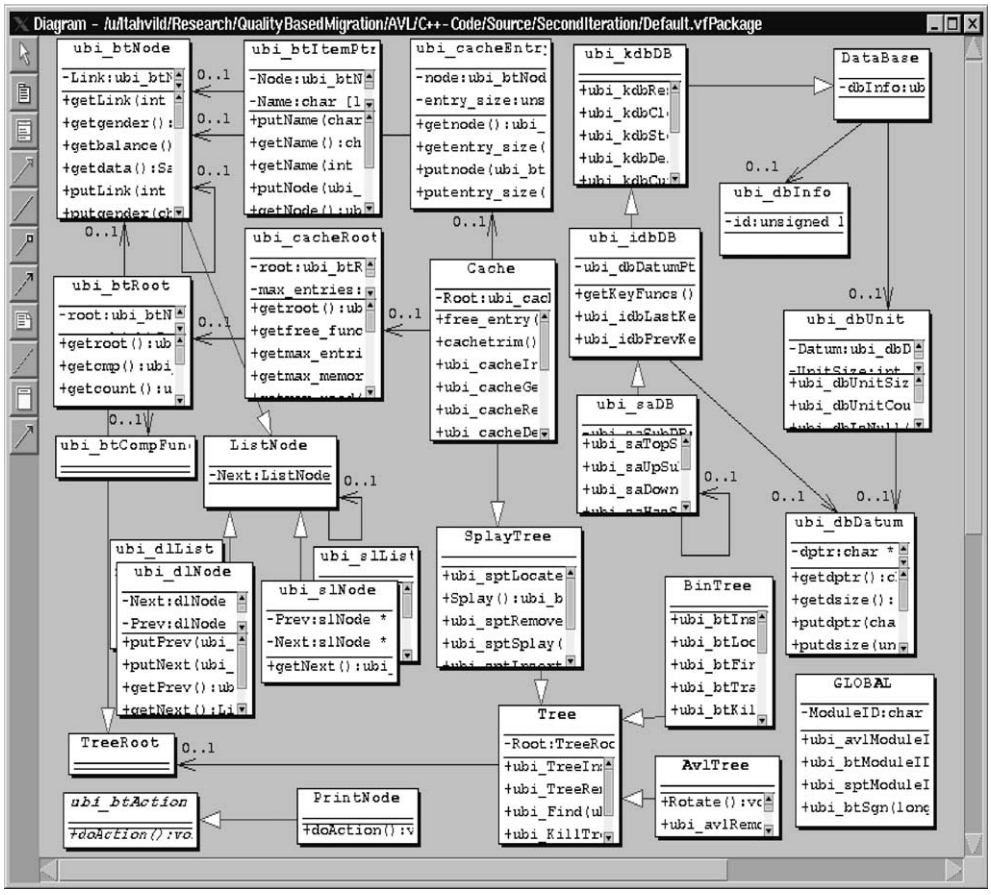

Fig. 11. Object model of AVL with design patterns.

Table 2
Impact of architectural level transformations on maintainability and performance

System	Pattern name	Perf. time (% diff)	Perf. Dhrystone (% diff)	SMI Method A (% diff)	SMI Method B (% diff)	SMI Method C (% diff)
AVL/WELTAB	Abstract Factory	−6.48	7.82	−4.78	−7.22	−2.09
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	Factory Method	7.04	7.50	−1.94	−3.70	−1.19
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

6.3.1. 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 a maintainability improvement at the level of 6% on average as shown in Table 2. Similarly, this pattern allows for performance enhancement as well, because it allows for explicit superclass references and simplifies component interfaces. Our experimental results confirm this result for an average performance increase of 5%.

In this context, the *Facade* design pattern is another design pattern for re-architecting a software system into subsystems and helping minimize the communication and data flow dependencies between subsystems. This pattern shields the subsystem components from the clients, thereby reducing the number of objects that clients deal with and consequently making subsystems such easier to use and maintain. For giving an example, consider Fig. 10 that describes the AVL system after simple object model extraction (without considering any design patterns) in comparison with the Fig. 11 that describes the same system after the selected design patterns have been applied. In the AVL system, there are different lists such as a single link list (“ubi_slList” class)

and a double link lists (''ubi_dlList'' class) as shown in Fig. 10. Structuring this part of system into subsystems helps to reduce its complexity. For achieving this goal, we can introduce a facade object that provides a single simplified interface to the more general facilities of the link lists. ''ListNode'' class in Fig. 11 provides a higher level interface that can shield clients from these classes and acts as a *Facade*. Results in Table 2 indicate an average maintainability increase of the caliber of 7.5%. This pattern also promotes a weak coupling between subsystems and its clients which reducing compilation dependencies and providing of almost 11.4% better performance on average. Table 2 summarizes all these results with respect to performance and maintainability enhancements.

6.3.2. 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.

One proper example of this pattern can be the ''Cache'' class in AVL system in Fig. 11. An aggregate object such as a ''cache'' can provide a way to access its elements (e.g., ''ubi_cacheRoot'' and ''ubi_cacheEntry'') without exposing its internal structure. This may result to a system that is more maintainable because of the simplification of the aggregate interface. Our experimental results indicate an average increase of the maintainability at the level of 7%. However, the pattern reduces the 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 want to perform operations on objects that compose containers. ''Visitors'' make 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 such an average improvement of 4.5% for all of the maintainability indices. ''Visitors'' help on applying operations to objects that do not have a common parent class. This has as a result a reduction to the traversal time and therefore it may be considered as a heuristic that improves performance as illustrated in Table 2.

6.3.3. 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 2. Meanwhile, the application of these patterns is shown to provide better performance at the range of almost 7.04% on average. On the other hand, 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.

An alternative way to parameterize a system relies mostly on object composition. Specifically, this category of transformations 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 Figs. 9 and 12 clarify the existence of this ''factory object''. Fig. 9 depicts the object model obtained without considering any design patterns (simple object model extraction), while Fig. 12 illustrates the same system after applying the selected design patterns.

For example, consider the ''REPORT'' class in Fig. 9 that aims to support multiple printed reports. Different reports have different appearances and headers for printing. To be portable across different report style sheets, an application should not be hard-coded for a particular report. We can solve this problem by defining an abstract ''ReportGen'' class that declares an interface for creating each kind of reports as shown in Fig. 12. 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) as illustrated in Table 2.

7. Related work

Software quality has been recognized to be an important topic since the early days of software engineering. Over the last 30 years, a number of researchers and practitioners alike have examined how systems achieve software quality requirements. Boehm in (Boehm et al., 1978) classified a number of software attributes such as flexibility, integrity, performance, maintainability, and so on. The International Organization for Standard-

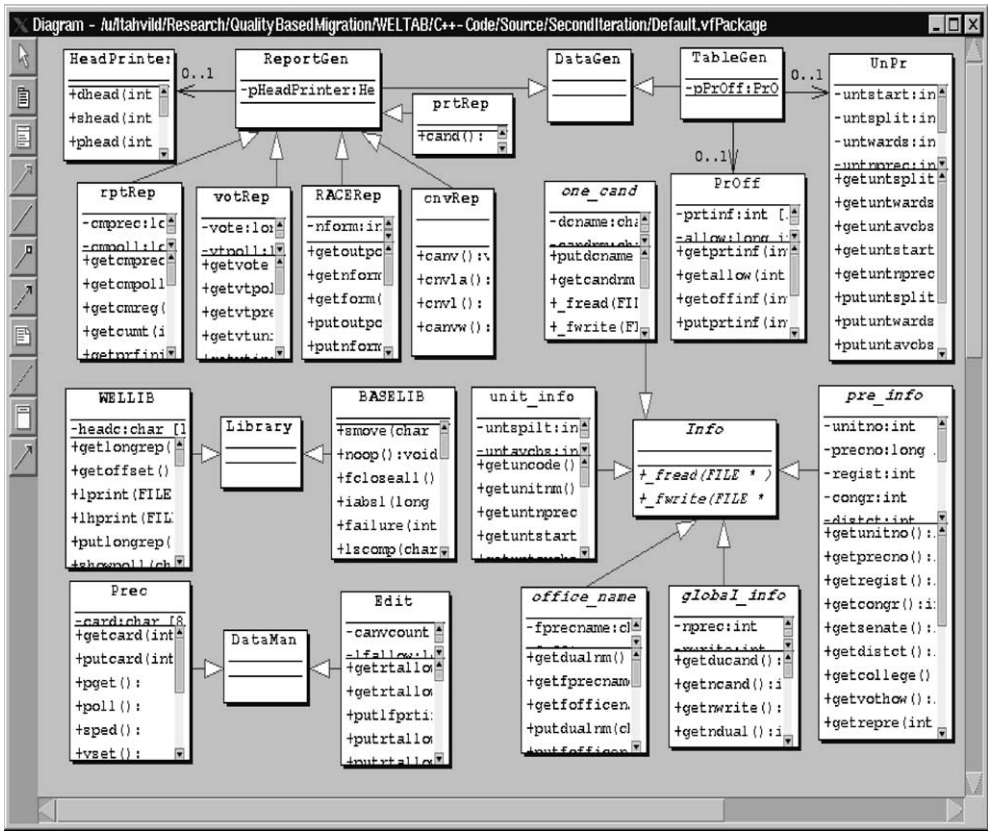

Fig. 12. Object model of WELTABIII with design patterns.

ization (ISO9126) introduced taxonomies of quality attributes (International Organization for Standardization (ISO), 1996) which divide quality into six characteristics namely: functionality, reliability, usability, efficiency, maintainability and portability. Complementary to the product-oriented approaches, the NFR Framework (Chung et al., 2000) takes a *process-oriented* approach to dealing with quality requirements. The NFR Framework is one significant step in making the relationships between quality requirements and design decisions explicit. The framework uses NFR to drive design to support architectural design level and to deal with the changes.

The recent interest on software architecture and design patterns have refocused attention on how these software qualities can be achieved (Klein et al., 1999; Kazman et al., 2000). Bergey has analyzed the relationship between software architecture and quality attributes (Bergey et al., 2000). Work conducted at the Software Engineering Institute (SEI) in Attribute-Based Architecture Style (ABAS) (Klein et al., 1999) was the first attempt to document the relationship between architecture and quality attributes. By codifying mechanisms, architects can identify the choices necessary to achieve quality attribute goals. This, in turn, will set a foundation for further software architectural design and analysis (Kazman et al., 2000).

As the re-engineering of legacy systems has become a major concern in today's software industry, most re-engineering efforts were focussed towards the analysis and migration of systems written in traditional programming languages such as Fortran, COBOL, and C (Baxter, 1990; Kontogiannis et al., 1998; Sneed, 1992). Unfortunately, none of them provides a re-engineering process that relates to the non-functional or quality requirement for the target system. The problem of coping with NFR during re-engineering has been experimentally tackled by developing a number of tools that met particular quality requirements. In (Finnigan et al., 1997), a tool-set has been developed that assists on the migration of PL/IX legacy code to C++ while maintaining comparable time performance. (Patil, 1999) 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 developed 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.

However, not much effort has been invested for systematically documenting quality attributes as a guide for the software re-engineering process at the architectural level. In this context, we believe that the proposed

re-engineering framework allows for specific quality requirements for the migrant system to be modeled as a collection of soft-goal graphs and for the selection of the transformational steps that need to be applied at the architectural (design) level of the legacy system being re-engineered.

8. Conclusions and future work

In this paper, we have presented a quality-driven framework for software re-engineering at the architectural level. The framework uses desirable qualities for the re-engineered code to define and guide the re-engineering process. The framework offers a workbench where 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 target system.

Specifically, the proposed framework addresses issues related to: (i) the design and development of a collection of comprehensive soft-goal interdependency graphs as they pertain to software qualities, such as performance and maintainability of large legacy systems, at the architecture level; (ii) the development of a comprehensive catalogue of transformations as these have been modeled in the soft-goal interdependency graphs and can be applied at the architecture level of a legacy system to address specific re-engineering objectives; (iii) the design and development of an analysis methodology that allows for the identification of error prone architectural design and source code programmatic patterns that can be "repaired" according to the transformational steps identified by the soft-goal graphs; (iv) the design and development of a prototype system that assists the re-engineering process and pertains to enhancements of the performance and maintainability of the migrant system.

Currently, we are working on extensions of the framework that allow for the estimation of the impact a transformation on maintainability and performance of a software system. We also aim to investigate algorithmic processes that can be used to automate the selection and application of transformations given a specific re-engineering scenario.

References

Aho, A.V., Sethi, R., Ullman, J., 1988. Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading, MA.

Baxter, I., 1990. Transformational maintenance by reuse of design histories. Ph.D. thesis, Department of Computer Science, University of California, Irvine.

Bergey, J., Barbacci, M., William, W., 2000. Using quality attribute workshops to evaluate architectural design approaches in a major system acquisition: A case study. Technical Report CMU/SEI-2000-TN-010, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

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

Chikofsky, E.J., CrossII, J.H., 1990. Reverse engineering and design recovery: A taxonomy. IEEE Software 7 (1), 13–17.

Chung, L.K., Nixon, B.A., Yu, E., Mylopoulos, J., 2000. Non-Functional Requirements in Software Engineering. Kluwer Publishing, Dordrecht.

Coleman, D., Lowther, B., Oman, P., 1995. The application of software maintainability models in industrial software systems. The Journal of Systems and Software 29, 3–16.

Cordy, J.R., Carmichael, I.H., 1993. The txl programming language syntax and semantics version 7. Technical Report 93-355 (June), Department of Computing and Information Sciences, Queen's University, Kingston, Canada.

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

Finnigan, P. et al., 1997. The software bookshelf. IBM Systems Journal 36 (4), 564–593.

Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading, MA.

Gamma, E., Helm, R., Jahnson, R., Vlissides, J., 1993. Design patterns: Abstraction and reuse of object-oriented design. In: Proceedings of the ACM 7th European Conference on Object-Oriented Programming, July 1993, pp. 406–431.

Gamma, E., Helm, R., Jahnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA.

Garlan, D., Shaw, M., 1993. An Introduction to Software Architecture. World Scientific Publishing Co., Singapore.

Garlan, D., Kaiser, G.E., Notkin, D., 1992. Using tool abstraction to compose system. IEEE Computer 25, 30–38.

GNU AVL Libraries, 1999. Also available at <http://www.interads.co.uk/~crh/ubiqx>.

Halstead, M., 1977. Elements of Software Science. Elsevier North-Holland Inc.

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

Kazman, R., Klein, M., Clements, P., 2000. Attam: Method for architecture evaluation. Technical Report CMU/SEI-2000-TR-004 ADA382629, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Klein, M., Bass, L., Kazman, R., 1999. Attribute-based architecture styles. Technical Report CMU/SEI-99-TR-022 ADA371802, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Kontogiannis, K., Martin, J., Wong, K., Gregory, R., Müller, H., Mylopoulos, J., 1998. Code migration through transformations: An experience report. In: Proceedings of IBM CASCON'98 Conference, 1998, pp. 1–13.

Lowther, B., 1993. The application of software maintainability metric models to industrial software systems. Master's thesis, Department of Computer Science, University of Idaho.

McCabe, T., 1976. A complexity measure. IEEE Transactions on Software Engineering 2 (4), 308–320.

Műller, H., Orgun, M., Tilley, S., Uhl, J., 1993. A reverse engineering approach to subsystem identification. Journal of Software Maintenance and Practice 5, 181–204.

Nixon, B.A., 1993. Dealing with performance requirements during the development of information systems. In: Proceedings of the IEEE International Symposium on Requirements Engineering, January 1993, pp. 42–49.

Oman, P., Hagemeister, J.R., 1994. Constructing and testing of polynomials predicting software maintainability. The Journal of Systems and Software 24, 251–266.

Parnas, D.L., 1972. On the criteria to be used in decomposing systems into modules. Communications of the ACM 15, 1053–1058.

Patil, P., 1999. Migration of procedural systems to object oriented architectures. Master's thesis, Department of Electrical and Computer Engineering, University of Waterloo.

Sneed, H., 1992. Migration of procedurally oriented cobol programs in an object-oriented architecture. In: Proceedings of the IEEE International Conference on Software Maintenance (ICSM), October 1992, pp. 105–116.

Software Refinery, Reasoning Systems. 1984. Also available at <http://www.reasoning.com>.

Tahvildari, L., Kontogiannis, K., 2000. A workbench for quality based software re-engineering to object oriented platforms. In: Proceedings of ACM International Conference in Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)—Doctoral Symposium, Minneapolis, MN, USA, October 2000, pp. 157–158.

Tahvildari, L., Kontogiannis, K., 2002. On the role of design patterns in quality-driven re-engineering. In: Proceedings of the IEEE 6th European Conference on Software Maintenance and Reengineering (CSMR), Hungary, Budapest, March 2002, pp. 230–240.

Tahvildari, L., Singh, A., 1999. Software bugs. In: Webster, J.G. (Ed.), Encyclopedia of Electrical and Electronics Engineering. John Wiley & Sons, New York, pp. 445–465.

Tahvildari, L., Gregory, R., Kontogiannis, K., 1999. An approach for measuring software evolution using source code features. In: Proceedings of the IEEE Asia-Pacific Software Engineering (APSEC), Takamatsu, Japan, December 1999, pp. 10–17.

Tahvildari, L., Kontogiannis, K., Mylopoulos, J., 2001. Requirements-driven software re-engineering. In: Proceedings of the IEEE 8th International Working Conference on Reverse Engineering (WCRE), Stuttgart, Germany, October 2001, pp. 71–80.

The Open Group Architecture Framework Version 7, 2001. Also available at <http://www.opengroup.org/public/arch>.

Together/C++ UML Editor. 1999. Also available at <http://www.togethersoft.com>.

Weicker, R.P., 1984. Dhrystone: A synthetic systems programming benchmark. Communications of the ACM 27 (10), 1053–1058.

WELTAB Election Tabulation System. 1980. Also available at <http://pathbridge.net/reproject/cfp2.htm>.

Ladan Tahvildari received her M.Sc. degree from the Department of Electrical and Computer Engineering at the University of Waterloo, Canada in 1999 where she is pursuing her Ph.D. degree. Her research interests include software evolution, program understanding, quality based re-engineering. She has been awarded an IBM Ph.D. Research Fellowship from the Center for Advanced Studies at IBM Canada, and also Ontario Graduate Scholarship (OGS) in Science and Technology from Government of Canada. She can be reached at ltahvild@swen.uwaterloo.ca.

Kostas Kontogiannis received his Ph.D. degree from McGill University in 1996, and is Associate Professor at the University of Waterloo, Department of Electrical and Computer Engineering. His research interests focus on the design and development of tools for software re-engineering with particular emphasis on software transformations, and legacy software migration to network-centric computing platforms. His publication list includes more than 50 refereed journal and conference proceedings papers. He was also the recipient of the 2001 IBM University Partnership Program Award. He can be reached at kostas@swen.uwaterloo.ca.

John Mylopoulos received his Ph.D. degree from Princeton University in 1970, and is Professor of Computer Science at the University of Toronto. His research interests include knowledge representation and conceptual modeling, covering languages, implementation techniques, and applications. His publication list includes more than 150 refereed journal and conference proceedings papers and six edited books. He is the recipient of the first ever Outstanding Services Award given out by the Canadian AI Society (CSCSI), a co-recipient of the best paper award of the 1994 International Conference on Software Engineering (ICSE), a fellow of the American Association for AI (AAAI), and is currently serving a 4-year term as president of the VLDB Endowment. He can be reached at jm@cs.toronto.edu.