A Methodology for Developing Transformations Using the Maintainability Soft-Goal Graph *

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

Abstract

Over the past few years, we are experiencing a surge of evolution problems related to legacy object-oriented systems. Object orientation provides means for systems to be well-designed to meet numerous non-functional quality requirements. However, mismanagement of the maintenance process and ill-planned modifications usually are amplified in an object-oriented context. The paper presents a re-engineering framework that defines and categorizes a collection of source code transformations that aim to introduce design patterns in an ill-structured object-oriented system due to excessive maintenance process. The framework allows for five categories of transformations to be defined and associated through soft-goal dependency graphs for the target system. A case study that illustrates the use of the framework for the restructuring and introducing of design patterns to the GNU AVL Library is presented.

1 Introduction

The re-engineering of legacy systems has become a major concern in today's software industry. Traditionally, most re-engineering efforts were focussed on systems written in traditional programming languages such as Fortran, COBOL, and C [1, 3, 20, 21, 27]. However, over the past few years we observe an increasing demand for the re-engineering of object-based systems. It is of no surprise that it becomes more and more difficult to maintain these object-oriented systems.

The re-engineering of object-oriented legacy systems requires a comprehensive framework to relate software transformation activities with specific requirements for the new target migrant system. We refer to this approach as "*Quality-Driven Object-Oriented Re-engineering*" [29, 32, 33]. More specifically, the major theme of the proposed approach is to exploit the synergy between *requirements analysis* [34], *software architecture* [13], and *reverse engineering* [6]. Understanding the architecture of an existing system aids in predicting the impact evolutionary changes may have on specific quality characteristics of the system [28]. Requirements analysis techniques, in turn, suggest what concepts are most useful in understanding how an existing system functions and in what manner it can evolve.

In previous work reported in [31], we proposed a layered software transformation re-engineering model for object-oriented systems that is driven by *maintainability* non-functional requirement for the target system, and is to be applied at the architectural level. In this paper, we are particularly interested to apply proper transformations on a target system as means to restructure the object-oriented legacy system so that the new migrant system conforms with specific design patterns and therefore possibly meets maintainability enhancement. The transformational steps are devised in a way that specific design decision are achieved. Such target design decisions are encoded as a *soft-goal graph* [7] and help guiding the of application of the transformation process.

This paper is organized as follows. Section 2 presents the conceptual model of the proposed transformations. Sections 3 discusses each transformation in details using descriptive notation. Section 4 associate the defined transformations to the maintainability soft-goal graph while Section 5 discusses how one can modify the soft-goal graph further by adding complex design pattern transformations. Section 6 discusses a case study using the proposed approach. Section 7 presents related work. Finally, Section 8 provides the conclusion.

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

2 Conceptual Model of the Transformations

It has been argued in the software engineering community that the use of design patterns [5, 12, 15, 16] has a positive effect on system qualities. The process of devising and composing transformations that introduce such design patterns in an ill-designed object-oriented system poses an investigating challenge for the re-engineering of such systems. The process is both a top-down for higher level transformations, and a bottom-up for the lower level design motifs. In this paper a transformation framework illustrated in Figure 1 is proposed.

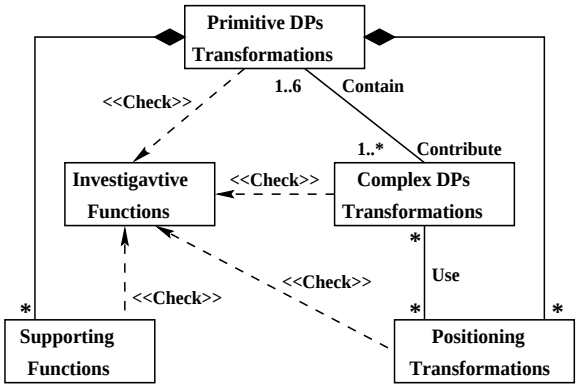

Figure 1. Meta-Model of the Transformations.

In defining a transformation, it is necessary to specify sets of preconditions and consequently assertions should be made about the program, such that a certain class exists or a given name-space is not already in use. For this purpose, we define a set of *investigative functions* to enable these assertions to be made as shown in Figure 1. Investigative functions serve two related roles. First, they are implemented as actual operations that can be applied to an object-oriented program to extract some information about the program. Second, they are used as predicates examining whether a specific transformation can be applied in a specific source code context. Examples include: i) to test whether a method is in a certain class, ii) to test whether there is a given class in the program.

In describing a transformation, it may be necessary to extract richer content from the program code than the information provided by the investigative functions. For this purpose, we define *supporting functions* as shown in Figure 1. For example, we may wish to build an interface from a class based on the signatures of its public methods. Supporting functions can be used to perform this type of task. We associate preconditions and postconditions that are implemented by the investigative functions as discussed above. The dependency relationship between these two classes is depicted in Figure 1. Examples from this category of functions include: i) construct and return an empty class, ii) construct and return an interface that reflects all the public methods of a given class. Investigative and supporting functions are proper functions without any functional side-effects on the program behavior.

A *positioning transformation* as shown in Figure 1 aims to introduce refactorings [22] to the original system towards achieving the desired target requirement (*e.g.*, enhance the maintainability). Most of them are standard and would be part of any refactoring suite [11, 22], such as *addClass* [11] operation.

In developing a transformation related to a particular design pattern, we aim to reuse previously defined transformations. For example, a class may register another class only via an interface (we call *ABSTRACTION*). These design motifs lead to *primitive design pattern transformations* as shown in Figure 1. Each of them is specified by describing how can be applied and what their effects may be.

- **Transformation Process**. This is a concise, step-by-step description on how to carry out and implement a transformation. The proposed transformation process is facilitated by the use of the framework illustrated in Figure 1. First, the assertions that must hold in order to be able to apply a transformation are presented. They pertain to the examination of the source code features that must be present for the transformation to be applied [28]. In defining these assertions, we use *investigative functions*. Second, the step-by-step of the implementation using supporting functions and positioning transformations are applied. Finally, specific conditions that must hold after a transformation is applied are evaluated using investigative functions.

- **Possible Effects on Soft-Goals**. The goal is to formalize and automate if possible the application of transformations that affect the specific target quality for the migrant system. This part associates each transformation to one or more soft-goals. In this paper, we consider only *maintainability* due to space limitation.

Consequently, the primitive design pattern transformations can be combined to produce *complex transformations* related to different design patterns as shown in Figure 1. We propose three ways to compose these transformations. One way is *sequencing* where transformations or refactorings are applied in order one after the other. The second way is *set iteration* where a transformation or refactoring is performed iteratively on a set of program elements. The third way is *concurrency* where a set of transformations are performed concurrently.

3 Primitive Design Pattern Transformations

As mentioned above, *primitive transformations* are design motifs that occur frequently. In this way, we consider them as lower level constructs in our framework. Each transformation in this category is denoted by a transformation process description and a description of its possible effect on target requirements.

3.1 ABSTRACTION Transformation

This transformation aims to add an interface to a class. This enables another class to take a more abstract view of the first class by accessing it via the newly added interface. It requires two parameters namely : i) the name of the class to be abstracted (*c*), and ii) the name of the new interface to be created (*newInterface*).

- **Transformation Process :** For the applicability of the transformation, we need first to evaluate preconditions in the source code using investigative functions as follows : 1) the class *c* exists and 2) no class or interface with the name *newInterface* exists. Then, the transformation entails the following steps : 1) an interface to be created using *abstractClass* supporting function that reflects the public methods of this class, 2) the addition of this interface to the program using positioning transformations such as *addInterface(newInterface)*, and 3) the addition of an *implements* link from the class to the newly created interface using a positioning transformation. Finally, the following conditions must hold after applying the transformation : 1) a new interface called *newInterface* exits, 2) the class *c* and the new interface have the same public interface, and 3) an *implements* link exists from the class *c* to the interface *newInterface*.
- **Possible effect on soft-goals :** High Control Flow Consistency(+), High Cohesion(++), High Data Consistency(++), Low I/O Complexity (−).

3.2 EXTENSION Transformation

This transformation aims to construct an abstract class from an existing class and to create an *extends* relationship between the two classes. It is related to ABSTRACTION transformation but rather than building a completely abstract interface from the class, it builds an abstract class where only certain specified methods are declared abstractly. This transformation requires three parameters namely : i) the name of the existing class (*oldClass*), ii) the name of the class to be created (*newClass*), and iii) the name of the methods to be abstracted (*abstractMethods*).

- **Transformation Process :** For the applicability of the transformation, we need first to evaluate preconditions in the source code features using investigative functions as follows : 1) no class or interface with the name *newClass* may exist, 2) the *oldClass* must exist, and 3) any fields used by methods that are to be pulled up must not be public. Then, the transformation requires for its application the following steps : 1) to create an empty class called *newClass* using the *emptyClass* supporting transformation, 2) to insert the newly created class into the inheritance hierarchy just above the *oldClass* using the *addClass* positioning transformation, 3) to add for each method in *abstractMethods* to this new class using *addMethod* positioning transformation, and 4) to move any methods not in *abstractMethods* from the *oldClass* to the newly created class using the *pullUpMethod* positioning transformation. Finally, these conditions must hold after the application of the transformation : 1) a new class called *newClass* exists, 2) the *oldClass* and its new superclass define precisely the same type, 3) all methods in *oldClass* not in *abstractMethods* are moved to the superclass, 4) any method in *abstractMethods* will have an abstract method declared in the class called *newClass*, and 5) any fields used by th moved methods are also moved to the superclass.
- **Possible effect on soft-goals :** High Control Flow Consistency(+), High Cohesion(++), High Module Reuse (++), Low Data Coupling(−).

3.3 MOVEMENT Transformation

This transformation aims to move parts of an existing class to a component class, and to set up a delegation relationship from the existing class to its component. This one requires three parameters namely : i) the name of the existing class (*oldClass*), ii) the name of the new class to be created (*newClass*), and iii) the name of the methods to be moved (*moveMethod*.

- **Transformation Process :** For the applicability of the transformation, we need to evaluate preconditions in the source code features using investigative functions as follows : 1) the *oldClass* must exist, 2) the name of the *newClass* must not be used, and 3) the methods to be moved must belong to the *oldClass*, Then, the transformation requires the following steps for its implementation : 1) an empty class to first be added to the program using *addClass* positioning transformation, 2) an exclusive component of this class to be added to the *oldClass*, 3) each method to be moved first to be "abstracted" using the *abstractMethod* supporting function, 4) at this point, the *moveMethods* po-

sitioning transformation may be invoked to move the method to the new class. Finally, these conditions must hold after applying the transformation: 1) a new class called *newClass* has been added to the program, 2) the class *oldClass* has a field called "movement", 3) all methods or fields defined directly or indirectly in *oldClass* that are used by a method in *moveMethod* are now public, 4) the given methods have been moved to the *newClass*, and 5) the *oldClass* delegates invocations of the moved methods to methods that exhibit the same behavior in the *newClass*.

- **Possible effect on soft-goals:** High Modularity(++), Low Control Flow Coupling(−), High Module Reuse(+).

3.4 ENCAPSULATION Transformation

This transformation aims to be applied when one class creates instances of another, and it is required to weaken the association between the two classes by packaging the object creation statements into dedicated methods. This transformation requires three parameters namely: i) name of the class to be updated (*creator*), ii) name of the product class (*product*), and iii) name of the new constructor method (*createProduct*).

- **Transformation Process :** For the applicability of the transformation, we need to evaluate preconditions in the source code features using investigative functions as follows: 1) the class *creator* exists and 2) the *creator* class defines no method called *createProduct* that have the same signature as a constructor in the class *product*. Then, the transformation requires the following steps to be implemented: 1) for every constructor in the *product* class, a new method called *createProduct* is created using *makeAbstract* supporting function which performs this construction and to be added to the *creator* class using the supporting functions, 2) all *product* objects created in the *creator* class are replaced with invocations of the appropriate *createProduct* method using a positioning transformation to replace the given object creation expression *e* with an invocation of the method *createProduct* using the same argument list. Finally, these conditions must hold after applying the transformation: 1) for every *product* object creation expression in the *creator* class, a method called *creatorProduct* that creates the same object is added to the *creator* class, and 2) every *product* object creation expression in the *creator* class that is not contained in a method called *createProduct* is deleted.

- **Possible effect on soft-goals:** Low Control Flow Coupling(+), High Data Consistency(−), High Encapsulation(++), Low Data Coupling(++).

3.5 BUILDRELATION Transformation

This transformation is appropriate when one class (*c1*) uses, or has knowledge of, another class (*c2*), and the relationship between the classes to operate in a more abstract fashion via an interface is required. This transformation requires four parameters namely: i) the name of the class to be used (*c2*, ii) the name of the super class (*c1*), iii) the name of the abstract interface to be used (*usedInterface*), and iv) the name of methods (*methodName*.

- **Transformation Process :** For the applicability of the transformation, we need to evaluate preconditions in the source code features using investigative functions as follows: 1) the interface *usedInterface* and the classes *c1* and *c2* exist, 2) an *implements* link exists from the class *c2* to the interface *usedInterface*, 3) any static methods in the *c2* class are not referenced through any of the object references to be updated, and 4) any public fields in the *c2* class are not referenced through any of the object references to be updated. Then, the transformation requires the following steps to be implemented: 1) to register each object reference in the class *c1* that is of the type *c2*, 2) to exclude any references that are contained in any method called *methodName*, 3) to modify their existing types from the class *c2* to the *usedInterface*. Finally, these conditions must hold after applying the transformation: 1) all references to the *c2* class in the *c1* class not in *methodName* have been changed to refer instead to the *usedInterface* and 2) the initial conjuncts of the precondition simply ensure that referenced classes and interface exist and have the proper relationship.

- **Possible effect on soft-goals:** High Control Flow Consistency(++), Low Control Flow Coupling (+), Low I/O Complexity (−).

3.6 WRAPPER Transformation

This transformation aims to "wrap" an existing receiver class with another class, in such a way that all requests to an object of the wrapper class are passed to the receiver object it wraps, and similarly any results of such requests are passed back by the wrapper. It requires two parameters namely: i) the name of a single receiver class or a set of receiver classes to be wrapped (*client*), ii) the name of an interface that reflects how the receivers are used in the client classes (*interfaceName*), and iii) the name of the wrapper class (*wrapperName*).

- **Transformation Process :** For the applicability of the transformation, we need to evaluate preconditions in the source code features using investigative functions

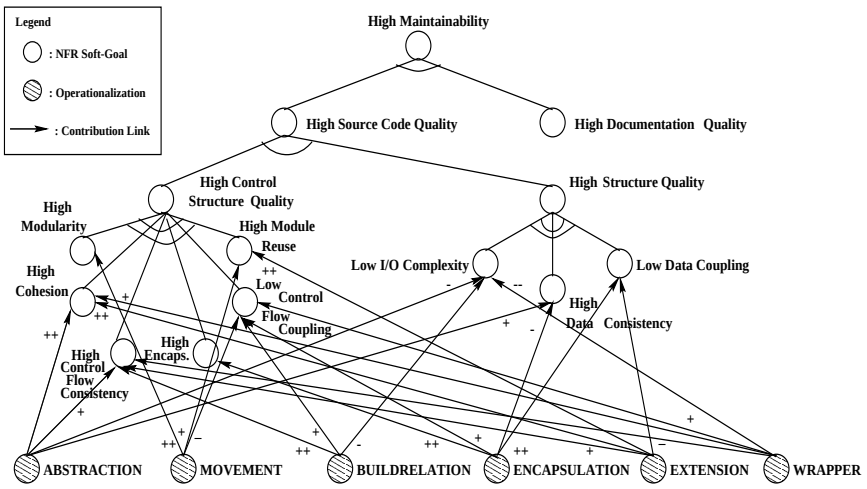

Figure 2. Relating Primitive Design Patterns Transformations to Maintainability Soft-Goal Graph.

as follows: 1) the given interface must exist and 2) the name for the new wrapper class is not in use. Then, the transformation requires the following steps to be implemented: 1) the wrapper class is created and added to the program and 2) the wrapper class is used to wrap each of the receiver classes and, consequently, any clients that use these receiver classes are updated to wrap each construction of a receiver class with an instance of the wrapper class. Finally, the following conditions must hold after applying the transformation: 1) the wrapper class has been added to the program, 2) all object references to receiver classes in *client* have been changed to *wrapperName*, and 3) all creations of receiver of objects in the *client* have been updated.

- **Possible effect on soft-goals:** High Cohesion (+), High Control Flow Consistency(+), Low Control Flow Coupling (+), Low I/O Complexity (−−).

4 Modifications on Maintainability Graph

While the maintainability soft-goal graph as presented in [32] provides specific interpretation of what the initial non-functional requirement (NFR) of "maintainability" means, it does not yet provide means for guiding the transformation process and actually achieving the desired quality. At some point, when the non-functional requirements have been sufficiently refined, one must be able to identify and associate actions for achieving these NFR (which are treated as NFR softgoals) and then assess the specific solutions for the target system. It is important to note that there is a "gap" between NFR softgoals and development techniques. This section associates primitive design pattern transformations with the maintainability soft-goal graph as shown in Figure 2. We call these associations *operationalizations* of the NFR soft-goals [7]. Like other softgoals, operationalizing softgoals makes a *contribution*, positive or negative, towards parent softgoals in terms of relations such as AND, OR, $+$, $++$, or $-$, $--$.

The proposed primitive design pattern transformations provide a body of knowledge to modify soft-goal dependency graph for maintainability as shown in Figure 2. For example, let us consider the challenge of achieving "High Cohesion" for a module in order to satisfy "High Maintainability" as the top level target goal. One possible alternative is to use the *ABSTRACTION* primitive design patterns transformation as shown in Figure 2. In this case, *ABSTRACTION* is a development technique or operationalization that can be implemented. It is a candidate for the task of meeting the high cohesion NFR as a positive positive contribution ($++$). This is contrasted with "High Cohesion", which is still a *software quality attribute*, *i.e.*, a non-functional requirement. We say that the *ABSTRACTION* transformation *operationalizes* high cohesion. We also say that the high cohesion NFR is *operationalized by ABSTRACTION* transformation. Operationalizing softgoals are drawn as filled circles and are just another type of soft-goal graph nodes.

5 Complex Design Pattern Transformations

In this section, we discuss how design patterns in the GoF book [12] can be defined as a composition of the primitive design pattern transformations that were discussed in Section 3. This further step also enable us to enhance the maintainability soft-goal graph in such a way that we have the target design decisions as leafs in the soft-goal graph as shown in Figure 3. Within the limit space of this paper,

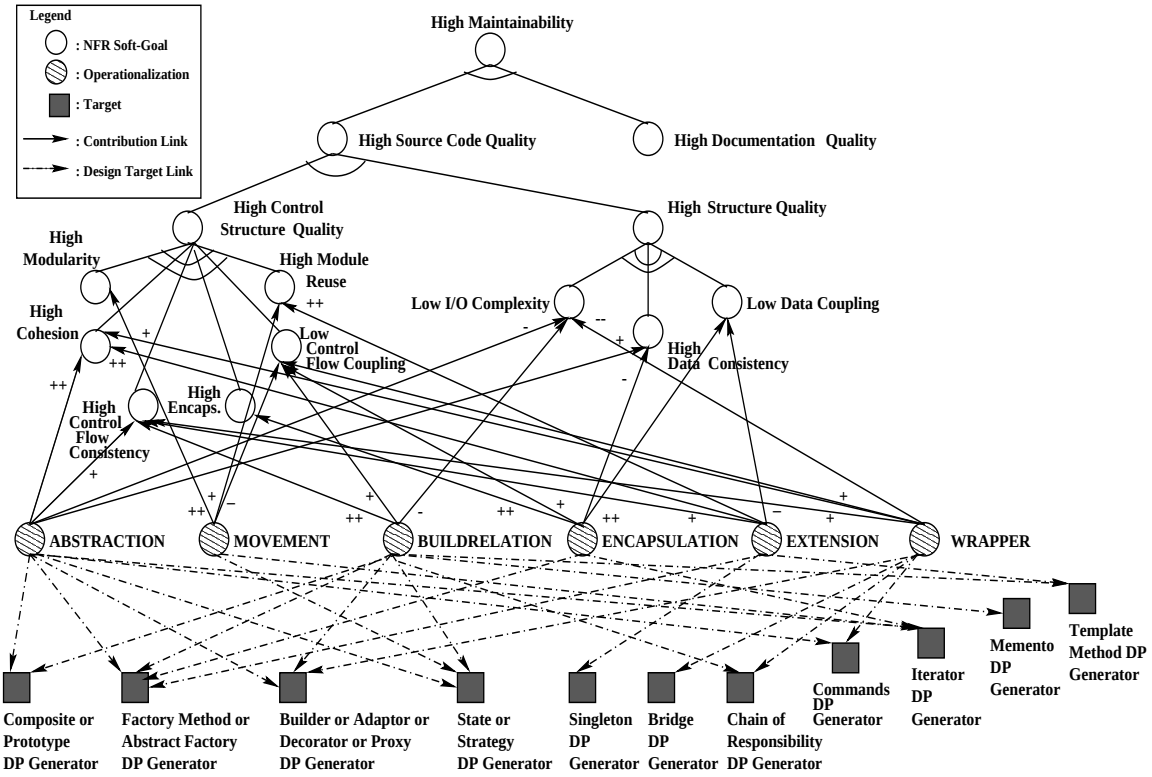

Figure 3. Relating Complex Design Patterns Transformations to Maintainability Soft-Goal Graph.

we present only the creation of a subset of the GoF patterns even though all of them are shown in Figure 3. Some of the fundamental and commonly used GoF patterns that we consider are the "Factory Method" from the *Creational Patterns* category, the "Composite" from the *Structural Patterns* category, and the "Iterator" from the *Behavioral Patterns* category. These are sufficiently complex to illustrate the use of the proposed transformation composition framework.

The intend of the Factory Method pattern is to define an interface for creating an object, but let subclasses decide which class to instantiate [12]. The *Factory Method Design Pattern Generator* lets a class defer its instantiation to subclasses. The transformation consists of the following steps: 1) the application of the "ABSTRACTION" primitive design pattern transformation to generate an interface that reflects how the creator class uses the instances of the product that it creates, 2) the application of the "ENCAPSULATION" primitive design pattern transformation so that the construction of product objects can be encapsulated inside dedicated, overridable methods in the creator class, 3) the application of the "BUILDRELATION" primitive design pattern transformation so that the creator class can register the product class only via the interface created in the previous step, and 4) the application of the "EXTENSION" primitive design pattern transformation so that the creator class can be inherited from an abstract class where the construction methods are declared abstractly.

The intend of the Composite pattern is to enable a client class to treat a single component object or a composition of objects in a uniform fashion [12]. The result of the *Composite Design Pattern Generator* transformation is that the client class uses the component class through its interface. It is also easy to extend the client so that it uses compositions of components in place of the single component instances. The transformation consists of the following steps: 1) the application of the "ABSTRACTION" primitive design pattern transformation on the component class in order to produce the component interface, and 2) the application of the "BUILDRELATION" primitive design pattern transformation in order to abstract the client class from the component class and use the component interface instead.

The intend of the Iterator pattern is to enable sequential access to the elements of an aggregate object without exposing the underlying representation of the object [12]. The *Iterator Design Pattern Generator* allows for multiple concurrent iterations over the aggregate object in a way that the underlying structure of the aggregation is not exposed. The transformation consists of the following steps: 1) the application of the "MOVEMENT" primitive design pattern

transformation to copy the iteration methods and fields to the new iteration class, which is parameterized with an instance of the aggregate class and delegates any internally generated, more iterator requests to this instance, 2) the application of the "ABSTRACTION" primitive design pattern transformation on the iterator class in order to produce an iterator interface, and 3) the application of the "ENCAPSULATION" primitive design pattern transformation to add an construction method for the iterator to the aggregate class.

6 A Case Study : GNU AVL Library

In this section, we discuss the usage of the proposed transformations towards the design and development of a quality and requirements-driven software re-engineering framework. We have applied this layered catalogue of transformations on the *GNU AVL Library* which is a public domain library written in C for sparse arrays, AVL, Splay Trees, and Binary Search Trees [14]. 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 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 6 source files and 3 library files.

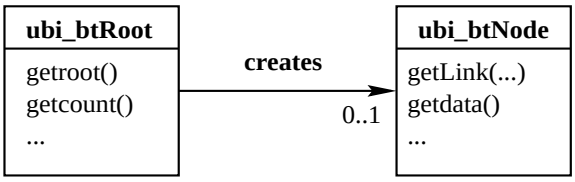

Figure 4. A Part of Object Model of AVL.

For this experiment, we have started from an object-oriented version of AVL Library [23] that was not structured and did not support design patterns for its implementation. Our objective is to transform this object-oriented system to a new design that conforms with specific design patterns and its maintainability characteristics are enhanced. For this task, we have first 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 [30]. The *Factory Method Generator* of this category defines an interface for creating an object but lets subclasses decide which class to instantiate. By selecting this design decision, we need to apply the four primitive design pattern transformations namely : ABSTRACTION, ENCAPSULATION, BUILDRELATION, and EXTENSION.

By analyzing the code and checking its features through investigative functions, it is concluded that there are two classes namely : *ubi_btRoot* and *ubi_btNode* that are of particular interest as shown in Figure 4 (because of space limitation, we illustrate the use of the layered architecture of transformations in a subset of the AVL library code). The ABSTRACTION transformation can be used to add an interface to *ubi_btNode*. This enables *ubi_btRoot* to take a more abstract view of this class by accessing it via this interface that is called *ListNode*. The effect of applying this transformation is depicted in Figure 5. *ListNode* interface has been added that provides an abstract view of the *ubi_btNode* class.

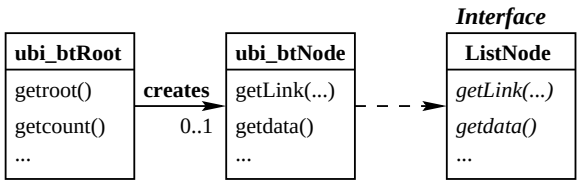

Figure 5. The Same Part of AVL After Applying ABSTRACTION Transformation.

The contributions of this transformation towards maintainability are denoted as follows :

$$\begin{aligned} AND(\quad & OR \quad (HighControlFlowConsistency(+), \\ & \qquad HighCohesion(++)), \\ & OR \quad (HighDataConsistency(++), \\ & \qquad LowI/OComplexity(-))). \end{aligned} \tag{1}$$

Now, we can apply the ENCAPSULATE transformation as *ubi_btRoot* creates instances of *ubi_btNode* and it is required to weaken the binding between two classes by packaging the object creation statements into dedicated methods. For each constructor of the *ubi_btNode* class, a method of the same signature has been added to the *ubi_btRoot* class that returns the same object as the corresponding constructor as shown in Figure 6. Also, all creations of *ubi_btNode* objects in the *ubi_btRoot* class have been updated to invoke these methods instead. The contributions of this transformation towards maintainability are denoted as follows :

$$\begin{aligned} AND(\quad & OR \quad (HighControlFlowConsistency(+), \\ & \qquad HighCohesion(++), \\ & \qquad LowControlFlowCoupling(+), \\ & \qquad HighEncapsulation(++)), \\ & OR \quad (HighDataConsistency(+), \\ & \qquad LowI/OComplexity(-), \\ & \qquad LowDataCoupling(++))). \end{aligned} \tag{2}$$

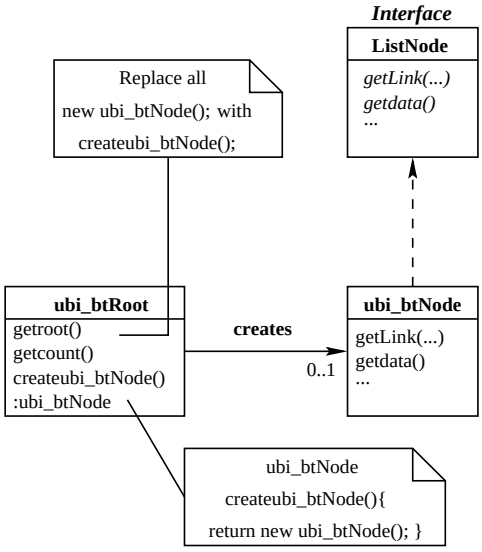

Figure 6. The Same Part of AVL After Applying ENCAPSULATION Transformation.

Then, we can apply the BUILDRELATION transformation as in the *ubi_btRoot* class all references to the *ubi_btNode* class should have been replaced by references to the *ListNode* interface as shown in Figure 7.

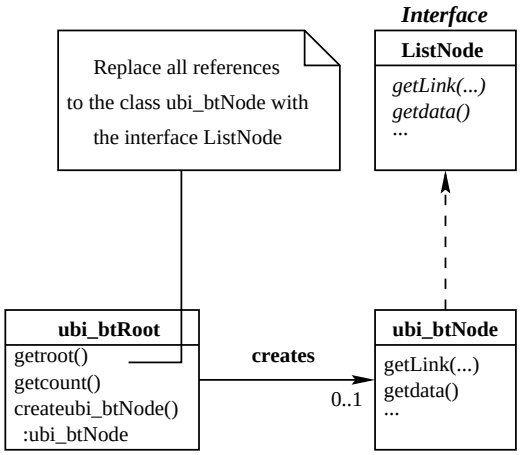

Figure 7. The Same Part of AVL After Applying BUILDRELATION Transformation.

The contributions of this transformation towards maintainability are denoted as follows:

$$
\begin{aligned}
AND(\quad & OR \quad (HighControlFlowConsistency(+++), \\
& \quad\;\; HighCohesion(++), \\
& \quad\;\; LowControlFlowCoupling(++), \\
& \quad\;\; HighEncapsulation(++)), \\
& OR \quad (HighDataConsistency(+), \\
& \quad\;\; LowI/OComplexity(--), \\
& \quad\;\; LowDataCoupling(++))). \qquad (3)
\end{aligned}
$$

The last step is the application of the EXTENSION transformation to construct an abstract class which is called *abs_ubi_btRoot* from *ubi_btRoot* and creates an *extends* relationships between these two classes. The original *ubi_btRoot* class simply inherits this class and provides definitions for the construction methods as shown in Figure 8.

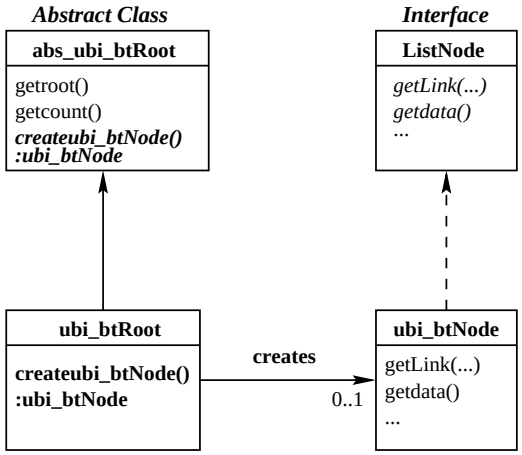

Figure 8. The Same Part of AVL After Applying EXTENSION Transformation.

Finally, the contributions of this transformation towards enhancing maintainability are denoted as follows:

$$
\begin{aligned}
AND(\quad & \\
& OR \quad (HighControlFlowConsistency(+++), \\
& \quad\;\; HighCohesion(+++), \\
& \quad\;\; LowControlFlowCoupling(++), \\
& \quad\;\; HighEncapsulation(++), \\
& \quad\;\; HighModuleReuse(++)), \\
& OR \quad (HighDataConsistency(+), \\
& \quad\;\; LowI/OComplexity(--), \\
& \quad\;\; LowDataCoupling(+))). \qquad (4)
\end{aligned}
$$

This last equation can be further simplified. The first OR denotes that after applying those four transformations, the target code has positive impact on High Control Structure Quality. However, the second OR denotes that after applying those transformations, High Structure Quality of code has not been changed. The AND contribution between these two sub-goals as shown in Figure guides the re-engineering activity towards a system which may be more maintainable.

7 Related Work

Software quality has been recognized to be an important topic since the early days of software engineering [24]. Over the past 30 years, a number of researchers and practitioners alike have examined how systems can meet specific software quality requirements [4, 17].

Complementary to the product-oriented approaches, the NFR (Non-Functional Requirements) Framework [7] 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 non-functional requirements to drive design to support architectural design level and to deal with the changes.

The recent interest on software architecture and design patterns has refocused the attention on how these software qualities can be achieved [18]. Klein and Barbacci have analyzed the relationship between software architecture and quality attributes [19, 2]. The Software Engineering Institute's (SEI's) work in Attribute-Based Architecture Style (ABAS) [19] 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.

The re-engineering of legacy systems has become a major concern in today's software industry. Traditionally, most re-engineering efforts were focused on systems written in traditional programming languages such as Fortran, COBOL, and C [20, 27]. Unfortunately, none of them provides means for guiding the re-engineering process within the context of achieving specific target qualities for the migrant system. The problem of coping with qualities or non-functional requirements during re-engineering has been experimentally tackled by developing a number of tools that met particular quality requirements [3, 10, 23].

Our idea on transformations which improve the design of the existing code builds upon the work of William Opdyke on refactoring C++ programs [22] where a suite of low-level refactorings that can be applied to a C++ program is proposed. This work was also used as the basis for the SmallTalk Refactoring [25]. Our work extends that prior work by using refactorings (positioning transformations) as a basis for developing a more sophisticated type of transformations that can introduce a design pattern and relate them to non-functional requirements to guide re-engineering activities.

Similarly, Eden [9] has developed a prototype tool called the *patterns wizard* that aims to apply a design pattern to an Eiffel program but it is not suitable for the re-engineering of legacy code. This work is very similar to ours in that it takes a meta-programming approach and organizes the transformations into four levels : design patterns (our complex design pattern transformations), micro-pattern (our primitive design pattern transformations), idioms (our positioning transformations), and the abstract syntax tree.

The works of Schulz [26] and Cinneide [8] are also related to the work presented in this paper. Specifically, in [26] the refactoring operations [22] were merged with the so-called *design operators*. However, in [8] the author merge refactoring work with a library of *mini-transformations*.

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, the proposed transformation framework allows for specific quality requirements for the migrant system to be modeled as a collection of soft-goal graphs. Moreover, it allows for the selection of transformations that need be applied at the architectural (design) level during the re-engineering process and towards achieving specific quality requirements.

8 Conclusion

We have presented an application of the proposed software transformation using maintainability soft-goal graph to support object-oriented software re-engineering at the architectural level. The framework enables for the transformations to be modeled in a language independent way. Also it enables for the reuse and composition of existing transformations.

We believe that 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.

Our current work involves applying this methodology to generate a broader variety of design patterns. Also, we work on extensions of the framework that allow for the estimation of the impact a transformation has on maintainability and other non-functional requirements (*e.g.,* performance) when applied to a software system. We are also investigating algorithmic processes as a constraint satisfaction problem that can be used to automate the selection and application of the transformations given specific re-engineering scenario.

For this work, we collaborate with the IBM Center for Advanced Studies at the IBM Toronto Laboratory and Consortium for Software Engineering Research.

References

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

[2] M. Barbacci, R. Ellison, J. Stafford, C. Weinstock, and W. Wood. Quality attribute workshops. Technical report cmu/sei-2001-tr-010, Software Engineering Institute, May 2001.

[3] I. Baxter and C. Pidgeon. Software change through design maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 250–259, October 1997.

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

[5] F. Buschmann et al. *Pattern-Oriented Software Architecture : A System of Patterns*. John Wiley and Sons, 1999.

[6] E. J. Chikofsky and J. H. CrossII. Reverse engineering and design recovery : A taxonomy. *IEEE Software*, pages 13–17, January 1990.

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

[8] M. O. Cinneide. *Automated Application of Design Patterns : A refactoring Approach*. PhD thesis, Department of Computer Science, Trinity College, Dublin, 2000.

[9] A. Eden, A. Yehudai, and J. Gil. Precise specification and automatic application of design patterns. In *Proceedings of the IEEE Automated Software Engineering (ASE)*, pages 143–152, November 1997.

[10] P. Finnigan et al. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, November 1997.

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

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

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

[14] Gnu avl libraries, 1999. Also available at http://www.interads.co.uk/˜crh/ubiqx.

[15] M. Grand. *Pattrens in Java*, volume 1. John Wiley & Sons, 1998.

[16] M. Grand. *Pattrens in Java*, volume 2. John Wiley & Sons, 1999.

[17] International organization for standardization (iso). Information Technology, Software Product Evaluation, Quality Characteristics and Guidelines for Their Use, ISO/IEC 9126, 1996.

[18] R. Kazman, L. Bass, G. Abowd, and M. Webb. Saam: A method for analyzing the properties of software architectures. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 81–90, May 1994.

[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. Műller, and J. Mylopoulos. Code migration through transformations : An experience report. In *Proceedings of IBM CASCON'98 Conference*, pages 1–13, 1998.

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

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

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

[24] R. S. Pressman. *Software Engineering : A Practitioner's Approach*. McGraw Hill, 2000.

[25] D. Roberts. *Eliminating Analysis in Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1999.

[26] B. Schulz, T. Genssler, B. Mohr, and W. Zimmer. On the computer aided introduction of design patterns into object-oriented systems. In *Proceedings of the* 27^{th} *TOOLS Conference*, 1998.

[27] H. Sneed and E. Nyary. Down-sizing large application programs. *Journal of Software Maintenance: Research and Practice*, 6(5):105–116, 1994.

[28] 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, Takamatsu, Japan, December 1999.

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

[30] L. Tahvildari and K. Kontogiannis. On the role of design patterns in quality-driven re-engineering. In *Proceedings of the IEEE* 6^{th} *European Conference on Software Maintenance and Re-engineering (CSMR)*, pages 230–240, Hungary, Budapest, March 2002.

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

[32] L. Tahvildari, K. Kontogiannis, and J. Mylopoulos. Requirements-driven software re-engineering. In *Proceedings of the IEEE* 8^{th} *International Working Conference on Reverse Engineering (WCRE)*, pages 71–80, Stuttgart, Germany, October 2001.

[33] L. Tahvildari, K. Kontogiannis, and J. Mylopoulos. Quality-driven software re-engineering. *The Journal of Systems and Software, Special Issue on : Software Architecture - Engineering Quality Attributes*, to appear.

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