A Software Transformation Framework for Quality-Driven Object-Oriented Re-engineering [*]

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

In re-engineering object-oriented legacy code, it is frequently useful to introduce a design pattern in order to improve specific non-functional requirements (e.g., maintainability enhancement). This paper presents a methodology for the development of a quality-driven re-engineering framework for object-oriented systems. First, a catalogue of design motifs (primitive design pattern transformations) is presented. Then, the transformations for the design patterns in the GoF book are defined as a composition of these primitive transformations. Non-functional requirements for the migrant system can be encoded using soft-goal interdependency graphs and can be associated with design pattern transformations that are applied for the migration of an object-oriented legacy system.

1 Introduction

Over the past few years, legacy system re-engineering has emerged as a business critical activity where the migrant system has to conform to hard and soft quality constraints (or non-functional requirements) such as "the new system should be more easily maintainable than the original system". In this paper, we propose a framework that enhances through re-engineering specific quality characteristics of a subject object-oriented system by transforming it to a form that possesses better maintainability characteristics. Specifically, we develop a catalogue of transformations that uses quality requirements to define and guide the re-engineering process.

[*]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.

We assume the following scenario. An existing object-oriented legacy system is being re-engineered to conform with a new requirement (*i.e.*,to enhance its maintainability). After analyzing the code and the requirements for the target system, it is concluded that the existing structure of the system makes the desired requirement difficult to achieve. It is therefore required to transform the system to a new structure where certain characteristics of its source code hold and facilitate further evolution activities. In this context, the introduction of design patterns has been attributed to reduce coupling and increase cohesion enabling thus certain types of program evolution to occur with minimal changes to the program itself.

In [29], we aimed 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. In [31], we included an initial compilation factors and simple transformations both at source code and architectural levels which affect two particular software qualities, performance and maintainability. We have also collected initial results using two medium-size software systems, mostly manually. We refer to this approach as "*Quality-Driven Software Re-engineering*" [32]. In [30], we examined design patterns from a different perspective namely, their classification and usage for software re-engineering and restructuring. Specifically, twenty three GoF design patterns were re-classified in terms of a layered model that is denoted by six different relations. We also discussed how the classification scheme can be useful for the re-engineering and restructuring of object-oriented systems. This classifications was developed to support the wider goal of improving the quality and the design of migrant code while maintaining its original functionality.

In this paper, we are interested to investigate transformations and refactoring operations that introduce GoF design

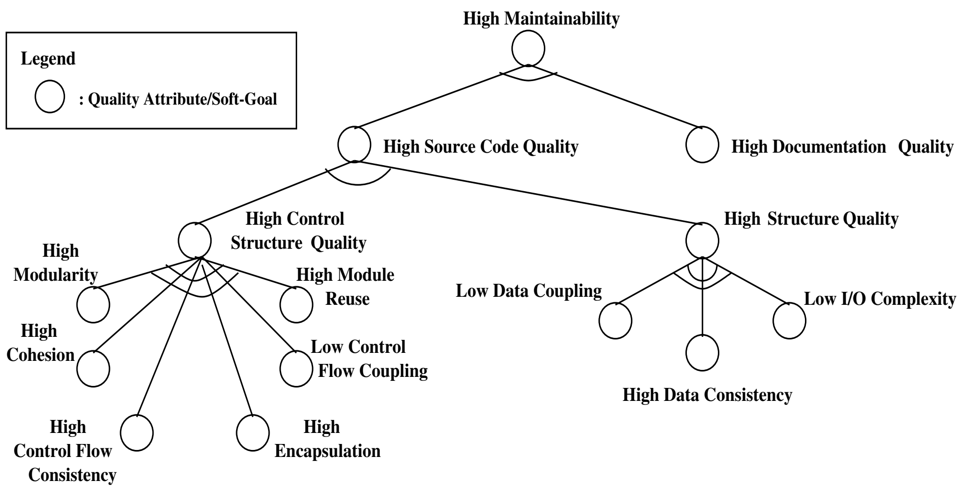

Figure 1. Maintainability Soft-Goal Interdependency Graph Decomposition.

patterns to an object-oriented system as means to restructure the system and for the new system to meet specific quality requirements. For achieving this goal, we first aim to develop a catalogue of transformations and refactoring operations [10] that can be used to enhance specific software qualities. Second, we aim to provide a process by which transformations can be composed and applied to a system through the introduction of design patterns. For the first objective, we present a comprehensive list of transformations that can be used to produce the design patterns presented in the GoF book. For addressing the second objective, we propose a layered model whereby the transformation process is based on investigative functions for evaluating preconditions, supportive functions for positioning the system to a form that a transformation can be applied, primitive design pattern transformations, and complex design pattern transformations.

This paper is organized as follows. Section 2 presents a soft-goal interdependency graph for maintainability and discusses its role in the proposed re-engineering context. Section 3 presents the proposed transformations, while Sections 4 and 5 discuss each transformation in detail using first-order predicate logic (FOPL) notation and also present their impact on soft-goal interdependency graph. Section 6 discusses an application scenario of the proposed transformations. Section 7 presents related work. Finally, Section 8 provides the conclusion and insights of future work.

2 Maintainability Soft-Goal Interdependency Graph

To represent information about software qualities, and the software transformations that may affect them, we adopt the NFR framework proposed in [6]. In the NFR framework, quality requirements are treated as potentially conflicting or synergistic goals to be achieved, and are used to guide and rationalize the various design decisions taken during system development. The NFR Framework introduces the concept of *soft-goals* whose level of success is evaluated by the success of other soft sub-goals. The *soft-goal interdependency graphs* have been proposed for supporting the systematic, goal oriented process of architectural design [6]. The leafs of the soft-goal interdependency graph represent transformations which fulfill or contribute positively/negatively to soft-goals above them. Given a quality constraint for a re-engineering problem, one can look up the soft-goal interdependency graph for that quality, and examine what are the transformations or soft-goals that may affect the desired quality positively or negatively.

In this paper, we have chosen maintainability enhancement as a re-engineering requirement for the new migrant system. Figure 1 shows portions of the soft-goal interdependency graph for the maintainability non-functional requirement. This graph attempts to represent and organize a comprehensive set of software attributes that relate to software maintainability. The graph was compiled after a thorough review of the literature [1, 4, 15, 17, 18, 22]. We have classified the maintainability non-functional requirements (NFR) soft-goal graph into two major areas namely, attributes that relate to source code quality [14], 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 a single arc. The source code quality soft-goal can be further decomposed into two sub-soft-goals namely, high control structure quality, and high information structure quality [14, 18] with an AND contribution as is depicted in Figure 1. It is important to note that in this work we only describe soft-goals relevant to the source code of the target system. It is also possible to identify maintainability related soft-goals that do not depend directly on source code properties. However, identifying such transformations would require knowledge about specific environmental factors (such as management and process modeling issues) and are outside the scope of the work presented in this paper.

For this research work, we are particularly interested to investigate proper transformations that introduce design patterns as means to restructure an object-oriented legacy system so that the new migrant system conforms with specific design patterns and therefore possibly meets specific non-functional requirement (NFR) criteria. It has been argued that maintainability can be partially achieved with the use of design patterns. For example, the *State* design pattern makes it easier to add new states in a system without altering the functionality of existing states. Ironically, experience drawn from software engineering practice reports that improved methods during system maintenance and evolution may result in higher maintainability indicators. Hence, it is important to consider how and by what means one can improve maintainability during re-engineering. For achieving this goal, we need to develop a list of transformations and refactoring operations [10] that introduce design patterns and also can be used to enhance specific software qualities during re-engineering. It is important to realize that these transformations can no longer be viewed as NFRs, since they can be implemented directly into a system. However, such transformations are still viewed as *soft-goals*, because can be decomposed if need be into more specific ones, and can be partially achieved.

3 A Methodology for Developing Transformations

In developing a transformation for a particular design pattern, we consider existing work in the design patterns and refactorings literature. examining the design pattern catalogue [5, 11, 12, 13], it is clear that certain motifs occur repeatedly across the catalogue. For example, a class registers another class only via an interface. It has been presented in [30] that specific design motifs can be combined in various ways to produce a wide variety of existing design patterns. The process of devising and composing transformations to introduce design patterns in an ill-designed object-oriented system is both a top-down one for higher level transformations, and a bottom-up one for the lower level design motifs. The importance of such techniques lies in the fact that they may allow us to implement a design pattern transformation as a composition of lower level design motifs. Our proposed transformation framework is defined in a layered model as illustrated in Figure 2. The rationale behind this proposed layered definition of transformations is further elaborated in the following sections.

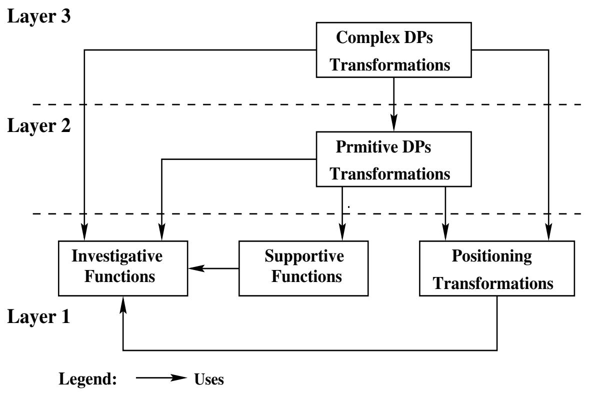
Layer 3
Complex DPs Transformations
Layer 2
Prmitive DPs Transformations
Investigative Functions
Supportive Functions
Positioning Transformations
Layer 1
Legend: Uses

Figure 2. Transformations in a Layered Model.

3.1 Transformation Notations

In defining a transformation, it is necessary to specify sets of preconditions for applying it and sets of postconditions to describe the effect of the transformation. In our work, we specify the preconditions and postconditions in first-order predicate logic (FOPL) with the following notation based on [20] and also used in [7, 25]. This will be used extensively wherever it will be necessary to be precise about the effect of applying a transformation to a program.

- $f[x/y]$: Denotes a function which maps the element x to y. This syntax is used in postconditions to denote the effect of the transformation or refactoring operations. Note that the name of a new function produced as the result of applying a transformation or refactoring is written with a prime ($'$), so stating that the function f is updated with the new element (x, y) would be written as : $f' = f[x/y]$.

- $\perp$: Used in a postcondition to denote an undefined value. For example, if a transformation removes a method called m, then to indicate that m no longer belongs to any class using the $classOf$ investigative function we assert that : $classOf' = classOf[m/\perp]$.

3.2 Proposed Transformation Framework

In developing a transformation related to a particular design pattern, we wish to reuse previously defined transformations. For example, a class may register another class

Name	Purpose
classCreation	A class that can be created by a given constructor.
classOf	A class to which the given method belongs.
containClass	A class that contains a given object reference.
containMethod	A method that contains a given object reference.
contstructorInv	A constructor that is invoked by a given object.
createsObject	True if a method creates an object of the same class.
declares	True if a class contains a method in its interface.
defines	True if a concrete method contains in a (super)class.
equalInterface	True if two interfaces declare the same public methods.
implmInterface	True if there is a link from an interface to another.
isAbstract	True if a class/method is declared to be abstract.
isClass	True if there is a given class in the program.
isInterface	True if there is a given interface in the program.
isPrivate	True if a method/field is a private member of its class.
isPublic	True if a method/field is a public member of its class.
isStatic	True if a method/field is a static member of its class.
name	Name of a given class/interface/method/constructor.
sig	Signature of a given method/constructor.
superclass	Direct superclass relationship of a class.
type	Returns the class/interface of a given reference.
uses	True if a method directly reference a field or invoked by another method or object reference.

Table 1. A List of Investigative Functions.

only via an interface (named *ABSTRACTION* in this research work). These design motifs lead to *primary design pattern transformations*as shown in layer 2 of Figure 2. These transformations can be combined in various ways to produce *complex transformations* related to different design patterns as shown in layer 3 of Figure 2. By focusing on the development of primitive design pattern transformations, we are able to build a library of useful transformations that can be reused. Each of the primitive design pattern transformation which will be further elaborated in Section 4 has the following structure :

Preconditions. These are assertions, written in first-order predicate logic, that must hold in order to be able to apply a transformation. They pertain to the examination of the source code features that must be present for the transformation to be applied [28]. In defining these preconditions, assertions should be made about the program, such as a certain class exists or a given name is not already in use. We need to define a set of *investigative functions* as shown in layer 1 of Figure 2 to enable these assertions to be made. Table 1 provides a brief description of these functions in an alphabetic order. These functions serve two related roles. First, they are implemented as actual operations that can be applied to an object-oriented program in order to extract information about specific source code features. Second, they are used as predicates for examining whether a specific transformation can be applied in a specific source code context (*e.g.*, to test whether a method is in a certain class or to find the signature of a given method).

Transformation Process. This is a concise, step-by-step description on how to carry out and implement design pattern transformations. The proposed transformation process is facilitated by the application of the following types of functions and transformations :

- *Investigative Functions*: aim to verify that specific conditions and source code properties hold before a transformation is attempted. Table 1 provides a list of such investigative functions.
- *Supportive Functions*: aim to position a system to a form that a specific transformation can be applied. These functions transform the system but they dot not necessarily contribute directly towards achieving the target requirement. Table 2 provides a list of such Supportive functions.
- *Positioning Transformations*: aim to introduce basic design patterns and refactorings [21] to the original system towards achieving the desired target requirement (i.e. enhance the maintainability). These operations are depicted in layer 1 of Figure 2. Most of them are standard and would be part of any refactoring suite [10, 21], such as the *addClass* [10] operation.

Postconditions. These pertain to mappings from investigative functions to investigative functions. They denote specific conditions that must hold after a transformation is applied.

Possible Effect on Soft-goals. The goal is to formalize and automate the application of transformations that affect the specific target quality for the migrant system. This part associates each transformation to one or more non-functional requirement. In this paper, we consider only *maintainability* just to illustrate the proposed quality-driven re-engineering framework.

4 Primitive Design Pattern Transformations

As mentioned above, *primitive transformations* and *refactorings* 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 precondition, a transformation process description, a postcondition, and a description of its possible effect on target requirements. These lower level transformations and refactorings are discussed in more detail in the following sections.

4.1 ABSTRACTION Transformation

This transformation is used to add an interface to a class. This enables another class to take a more abstract view of

Name(Parameters)	Informal Description	Preconditions	Postconditions
abstractClass (c, newClass)	Construct and return an interface (newClass) that reflects all the public methods of a given class(c).	1) $isClass(c)$	1) $isInterface' = isInterface[interface/true]$ 2) $equalInterface' = equalInterface[(c, interface)/true]$ 3) $name' = name[interface/newClass]$
abstractMethod (m)	Construct and return a method that has the same name & signature as a given method(m).	1) $isMethod(m)$	1) $isAbstract' = isAbstract[method/true]$ 2) $name' = name[method/name(m)]$ 3) $sig' = sig[method/sig(m)]$
emptyClass (name)	Construct and return an empty class(name).	No conditions	1) $name' = name[returned/name]$ 2) $\forall e : Method/Field/Constructor, \neg classOf(e) = returned$
makeAbstract (c, newClass)	Returns a method(newClass) creates the same object as a constructor(c).	No conditions	1) $createsObject' = createsObject[(c, returned)/true]$ 2) $name' = name[returned/newClass]$

Table 2. A List of Supportive Functions.

the first class by accessing it via the 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*).

- **Preconditions :**
  (a) $isClass(c)$
  (b) $\neg isClass(newInterface) \wedge \neg isInterface(newInterface)$
- **Transformation Process :** This transformation entails the following steps : 1) an interface (*tmp*) do be created using *abstractClass* supportive function that reflects the public methods of this class, 2) the addition of this interface to the program using positioning transformations such as *addInterface* and 3) the addition of an *implements* link from the class to the newly created interface.
- **Postconditions :**
  (a) $name' = name[tmp/newInterface]$
  (b) $isInterface' = isInterface[tmp/true]$
  (c) $equaleInterface' = equaleInterface[(c, tmp)/true]$
  (d) $implmInterface' = implmInterface[(c, tmp)/true]$
- **Possible Effect on Soft-goals :** High Control Flow Consistency(+), High Cohesion(++), High Data Consistency(++), Low I/O Complexity $(-)$

4.2 EXTENSION Transformation

This transformation is used 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 (*c*), ii) the name of the class to be created (*newClass*), and iii) the name of the methods to be abstracted (*abstractMethod*).

- **Preconditions :**
  (a) $\neg isClass(newClass) \wedge \neg isInterface(newClass)$
  (b) $isClass(c)$
  (c) $\forall f : Field,\ m : Method,\ f \in c,\ m \in c,\ m \notin abstractMethod,\ if\ uses(m, f)\ then\ \neg isPublic(f)$
- **Transformation Process :** This transformation requires for its application the following steps : 1) to create an empty class called *newClass* using the *emptyClass* supportive function, 2) the insertion of the newly created class into the inheritance hierarchy just above the class *c* using the *addClass* positioning transformation 3) to create an abstract method for each method in *abstractMethod* using positioning transformation which is added to this new class using *addMethod* positioning transformation 4) to move any methods not in *abstractMethod* from the class *c* to the newly created class using the *pullUpMethod* positioning transformation.
- **Postconditions :**
  (a) $isClass' = isClass[newClass/true]$
  (b) $equaleInterface' = equalInterface [(c, superclass'(c))/true]$
  (c) $\forall m : Method,\ m \in c,\ m \notin abstractMethod,\ classOf' = classOf [m/superclass'(c)]$
  (d) $\forall m : Method,\ m \in abstractMethod,\ declares' = declares [(superclass(c), m, direct)/true]$
  (e) $\forall m : Method,\ m \in c,\ m \notin abstractMethod,\ \forall f : Field,\ f \in c,\ uses(m, f),\ classOf' = classOf[f/superclass(c)]$

- **Possible Effect on Soft-goals:** High Control Flow Consistency(+), High Cohesion(++), High Module Reuse (++), Low Data Coupling(−).

4.3 MOVEMENT Transformation

This transformation is used 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 (*c*), and ii) the name of the new class to be created (*newClass*), and iii) the name of the methods to be moved (*moveMethods*).

- **Preconditions:**
  (a) $isClass(c)$

  (b) $\neg isClass(newClass) \wedge$
  $\neg\ isInterface(newClass)$

  (c) $\forall\ m\ \in\ moveMethods,\ m\ \in\ c$

  (d) $\forall\ f : Field,\ f\ \in\ c,$
  $name(f)\ \notin\ \text{``}movement''$

  (e) $if\ f : Field\ \in\ cls,\ cls\ \in\ superclass(c),$
  $name(f)\ =\ \text{``}movement''\ then\ isPrivate(f)$

- **Transformation Process:** This 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 *c* class, 3) each method to be moved first to be "abstracted" using the*abstractMethod* supportive function, 4) at this point, a proper positioning transformation may be invoked to move the method to the new class.

- **Postconditions:**
  (a) $isClass'\ =\ isClass[newClass/true]$

  (b) $\exists\ f : Field,\ f\ \in\ c\ such\ that$
  $type'\ =\ type[f/newClass]$
  $name'\ =\ name[f/\text{``}movement'']$

  (c) $\forall\ m : Method\ \in\ moveMethods,$
  $\forall\ x : Field/Method,\ defines(c,\ x),$
  $uses(m,\ x),\ isPublic'\ =\ isPublic[x/true]$

  (d) $\forall\ m : Method\ \in\ moveMthods,$
  $classOf'\ =\ classOf[m/newClass]$

  (e) $\forall\ m : Method\ \in\ moveMethods,$
  $\exists\ n : Method,\ classOf'(n)\ =\ c,$
  $name'(n)\ =\ name(m),$
  $sig'(n)\ =\ sig(m)\ such\ that$
  $uses'\ =\ uses[(n,m)/true]$

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

4.4 ENCAPSULATION Transformation

This transformation is used 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 can be implemented with 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*).

- **Preconditions:**
  (a) $isClass(creator)$

  (b) $\forall\ c : Constructor, c \in product$
  $\neg define(creator, createProduct,\ sig(c))$

- **Transformation Process:** This transformation requires the following steps to be implemented: 1) for every constructor in the *product* class, a new method called *createProduct* using *makeAbstract* supportive function that performs this construction and to be added to the *creator* class using the supportive function *addMethod*, 2) for all *product* objects created in the *creator* class to be replaced with invocations of the appropriate *createProduct* method. This last part requires a positioning transformation to replace the given object creation expression *e* with an invocation of the method *createProduct* using the same argument list.

- **Postconditions:**
  (a) $\forall\ e : ObjectCreationExprn,$
  $classCreation(e) = product,$
  $containClass(e)\ =\ creator,$
  $\exists\ m : Method\ such\ that$
  $createsObject'\ =\ createsObject$
  $[(constructorInv(e), m)/true]$
  $name'\ =\ name[m/createProduct]\ \wedge$
  $defines'\ =\ defines[(creator, m)/true]$

  (b) $\forall\ e : ObjectCreationExprn,$
  $classCreation(e)\ =\ product,$
  $containClass(e)\ =\ creator,$
  $name(containMethod(e))\ \neq$
  $createProduct,$
  $containMethod'\ =\ containMehod[e/\ \perp]$

- **Possible Effect on Soft-goals:** Low Control Flow Coupling(+), High Data Consistency(−), High Encapsulation(++), Low Data Coupling(++)

4.5 BUILDRELATION Transformation

This transformation is used 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. It may well happen that there are methods in the *c1* class that need to access the *c2* class directly. For example, they may instantiate the *c2* class, and these methods should be excluded from the transformation. 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*.

- **Preconditions :**
  (a) $isInterface(interface) \wedge isClass(c1) \wedge isClass(c2)$
  (b) $implmInterface(c2, usedInterface)$
  (c) $\forall\, m : Method,\ m \in c,\ isStatic(m),$ $\forall\, o : ObjectRef,\ type(o) = c2,$ $containClass(o) = c1,\ \neg\ uses(o, m)$
  (d) $\forall\, f : field,\ f \in c2,\ isPublic(f)$ $\forall\, o : ObjectRef,\ type(o) = c2,$ $containClass(o) = c1,\ \neg\ uses(o, f)$

- **Transformation Process :** This 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 interface *usedInterface*. This last step requires a positioning transformation to replace the given object reference *o* with the interface *usedInterface*.

- **Postconditions :**
  (a) $\forall\, o : ObjectRef,\ type(o) = c2,$ $containClass(o) = c1,$ $name(containMethod(o))\ in\ methodName,$ $type' = type[o/usedInterface]$

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

5 Complex Design Pattern Transformations

In this section we discuss how design patterns in the GoF book [11] can be defined as a composition of the primitive design pattern transformations which were discussed above in Section 4. Within the limits of this paper, we present only the creation of a subset of the GoF patterns. Some of the fundamental and commonly used GoF patterns we consider are the "Factory Method" from *Creational Patterns* category, the "Composite" from *Structural Patterns* category, and the "Iterator" from *Behavioral Patterns* category. These are sufficiently complex to illustrate the use of the proposed transformation composition framework. Figure 3 depicts the level of reuse of the primitive transformations for these three design patterns of the Gamma *et al* catalogue. As it can been seen, a considerable level of reuse was achieved. As shown in Figure 3, when a primitive transformation (also is called *operationalization* which is a possible design alternative for meeting NFRs in the target system [6]) makes a contribution towards one or more parent soft-goals, it is related to the latter in terms of a link labeled +, ++, or −, −−. A simple example of this interdependency graph is that the transformation "*Factory Method Design Pattern Generator*" contributes very positively (++) to the soft-goal *high control flow consistency* and negatively (−) to the soft-goal *low I/O complexity* using "BUILDRELATION" primitive design pattern transformation (a type of operationalization which can be implemented directly).

The intend of the Factory Method pattern is to define an interface for creating an object, but let subclasses decide which class to instantiate [11]. 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, 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 [11]. The result of *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, 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 [11]. The *Iterator Design Pattern Generator* allows for multiple concurrent iterations over the aggregate object in a way that

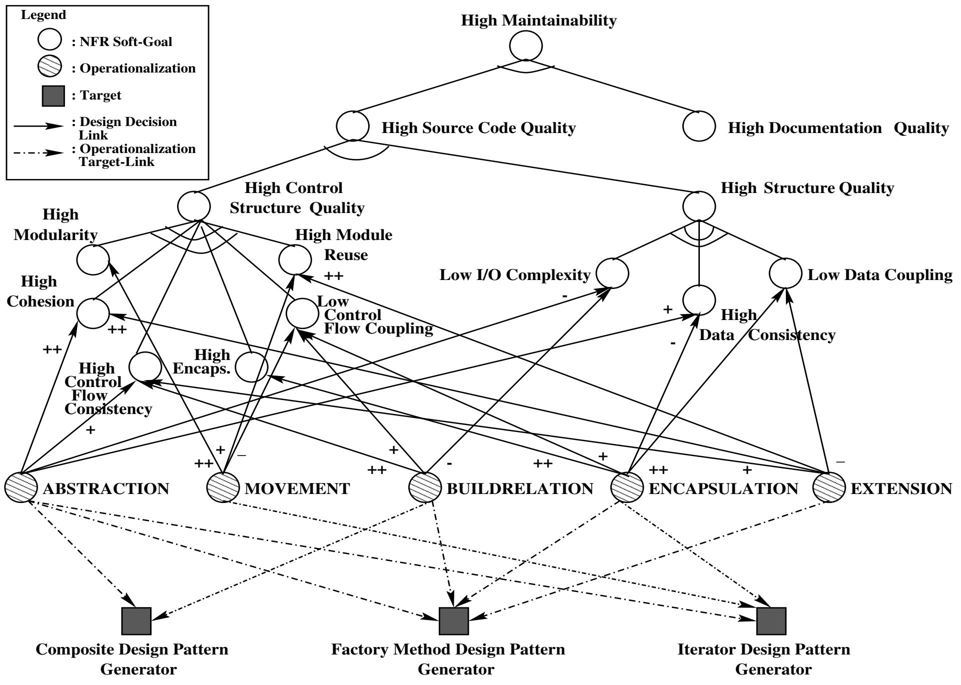

Figure 3. Relating Transformations to Maintainability Soft-Goal Graph.

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, 3) the application of the "ENCAPSULATION" primitive design pattern transformation to add an construction method for the iterator to the aggregate class.

6 Usage of the Transformation Framework

In this section, we discuss the usage of the proposed layered 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 *WELTAB Election Tabulation System* [33], a medium-size software system that supports the collection, reporting, and certification of election results by city and country clerk offices in the USA. WELTAB was originally written in an extended version of Fortran, then converted to C. An Object-Orientation Migration Tool [23] has been applied to migrate the C source code to new object-oriented $C++$ code.

For this experiment, we have started from an object-oriented version of WELTAB that was not structured and did not support design patterns for its implementation. Our objective was to transform this object-oriented WELTAB system to a new design that conforms with specific design patterns and its maintainability characteristics are enhanced. For this task, we have first considered *Structural Patterns* because they are concerned with the way classes and objects are composed to form larger structures. The *Composite* pattern describes how to build a class hierarchy that is made up of different kinds of objects. For example, in the WELTAB system, there are two classes, namely "RECORD" which produces base tables and, "REPORT" which prints the tables. It shows the necessity of having an abstract class that makes up these different kind of objects. The key point is to have a *Composite* pattern, namely "DateGen" that represents both primitives and their containers. We compose two objects into tree structures to represent part-whole hierarchy. This lets the clients "ReportGen" and "TableGen" treat individual objects and compositions of objects uniformly. This *Composite* pattern is found to improve maintainability because it allows for component sharing [30].

As for the introduction of *Behavioral Patterns*, we have considered the *Iterator* pattern which 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. Our experimental results [31] indicate that by applying the *Iterator* pattern, we obtain an increase in maintainability. One example for the introduction of this pattern is the "Info" class that provides a single simplified interface for accessing and traversing elements related to classes "VOTEUNIT", "OFFICE", "CMPREC", and "RECORD" generated as models for units, offices, precincts, and candidates. Structuring this part of the system into subsystems has been proven to reduce its complexity [31].

In this context, the proposed transformation scheme helped us to select those target design patterns that may have a significant measurable impact with respect to maintainability enhancements on the migrant code. These enhancements can be measured in term of software maintainability index metrics measurements.

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 [3, 15]. Complementary to the product-oriented approaches, the NFR (Non-Functional Requirements) Framework [6] 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 [16]. Klein and Barbacci have analyzed the relationship between software architecture and quality attributes [18, 1]. The Software Engineering Institute's (SEI's) work in Attribute-Based Architecture Style (ABAS) [18] 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 [19, 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 [2, 9, 23].

Our idea on transformations which improve the design of the existing code builds upon the work of William Opdyke on refactoring C++ programs [21]. He developed a suite of low-level refactorings that can be applied to a C++ program. This work was also used as the basis for the SmallTalk Refactoring [25]. Our work extends that 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 tasks.

Similarly, Eden [8] 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 [7] are also related to the work presented in this paper. Specifically, in [26] the refactoring operations [21] were merged with the so-called *design operators*. However, in [7] 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 as a layered architecture allows for achieving specific quality requirements in the migrant system to be modeled as a collection of soft-goal graphs.

8 Conclusion

We have proposed a layered software transformation framework to support object-oriented software re-engineering at the architectural level. The layered 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 that can be used to automate the selection and application of the transformations given specific re-engineering scenario.

References

[1] 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.

[2] 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.

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

[4] B. W. Boehm and H. In. Identifying quality requirement conflicts. *IEEE Software*, 13(2):25–35, March 1996.

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

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

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

[8] 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.

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

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

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

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

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

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

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

[16] 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.

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

[18] 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.

[19] 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.

[20] J. Loeckx and K. Sieber. *Foundation of Program Verification*. Wiley & Sons, 1987.

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

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

[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)*, Hungary, Budapest, March 2002.

[31] 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.

[32] 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.

[33] Weltab election tabulation system. Also available at http://pathbridge.net/reproject/cfp2.htm.