JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2004; **16**:331–361 (DOI: 10.1002/smr.299)

Research

Improving design quality using meta-pattern transformations: a metric-based approach

Ladan Tahvildari*,† and Kostas Kontogiannis

Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

SUMMARY

Improving the design quality of large object-oriented systems during maintenance and evolution is widely regarded as a high-priority objective. Furthermore, for such systems that are subject to frequent modifications, detection and correction of design defects may easily become a very complex task that is even not tractable for manual handling. Therefore, the use of automatic or semi-automatic detection and correction techniques and tools can assist reengineering activities. This paper proposes a framework whereby object-oriented metrics can be used as indicators for automatically detecting situations for particular transformations to be applied in order to improve specific design quality characteristics. The process is based both on modeling the dependencies between design qualities and source code features, and on analyzing the impact that various transformations have on software metrics that quantify the design qualities being improved. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: software reengineering; object-oriented metrics; program transformation; design flaws; non-functional requirements

1. INTRODUCTION

It has been widely argued both in research and industry that design defects cause systems to exhibit low maintainability, low reuse, high complexity and faulty behavior. Specifically, for object-oriented (OO) legacy systems which have been subjected to frequent modifications, detection and correction of such design flaws may become a complex task.

*Correspondence to: Ladan Tahvildari, Department of Electrical and Computer Engineering, University of Waterloo, 200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1.
†E-mail: ltahvild@uwaterloo.ca

Contract/grant sponsor: Natural Sciences and Engineering Research Council (NSERC) of Canada
Contract/grant sponsor: IBM Canada Ltd. Laboratory, Toronto - Center for Advanced Studies (CAS)

Received 7 June 2003
Revised 29 October 2003
Accepted 16 February 2004

Our previous work on improving the quality of OO legacy systems includes: (i) using metrics for quality estimation [1,2]; and (ii) proposing a software transformation framework that is based on soft-goal dependency graphs to enhance quality [3,4]. Both aspects have mostly been treated independently from each other. A natural extension to these efforts is to analyze the interaction of particular transformations and metrics in a systematic manner in order to suggest the use of transformations that may be helpful in improving quality as estimated by various metrics. In this paper, we first present a catalogue of OO software metrics that are related to OO design properties and to potential design flaws. Then, we present a framework of transformations that aims to improve error-prone design properties and to assist in enhancing specific qualities of a software system.

This paper is organized as follows. Section 2 proposes a classification of OO design flaws which is a step towards discovering recurring detection and correction methods. Section 3 discusses a quality-driven framework for software engineering, while Section 4 presents how meta-pattern transformations as design motifs can be modeled in soft-goal interdependency graphs (SIGs). Section 5 presents the proposed reengineering strategy using OO metrics to detect and correct design flaws, while Section 6 classifies a selection of OO metrics which are of the interest for this research. Section 7 discusses the impact of specific transformations on software features and metrics, while Section 8 proposes a model for a SIG representation. Section 9 presents a transformation selection and application algorithm. Section 10 discusses the architectural design and implementation issues for building the quality-driven object oriented reengineering (QDR) framework as a prototype which offers an interactive and incremental environment for improving the quality of an OO system based on the proposed approach. Section 11 presents a set of case studies related to the proposed classification and an application scenario of such case studies is further elaborated. Section 12 discusses the related work to this research and, finally, Section 13 provides the conclusion and insights of future work.

2. A CLASSIFICATION OF OO DESIGN FLAWS

In this section, we propose a classification of OO design flaws which is a step towards discovering recurring software design detection and correction methods.

Design properties are tangible concepts that can be directly assessed by examining the internal and external structure, relationships, and functionality of the system components, attributes, methods, and classes. An evaluation of the class definition and its external relationships (inheritance type) with other classes, as well as the examination of its internal components, attributes, and methods, can be used to reveal significant information that objectively captures the structural and functional characteristics of a class and its elements.

Overall, the design of a system could deteriorate for several reasons. Even though a class may have been designed taking into account all the principles of OO design (i.e., encapsulation, information hiding, data abstraction) in its initial stages of development it may lose its integrity due to: (i) extensive maintenance due to the addition of methods/data members; (ii) addition of excessive functionalities as a base class trying to accomplish too much for its derived classes; (iii) designs that aim to handle too many different situations, grouping what should be several different derived classes into a single class; (iv) excessive numbers of relationships and associations with other classes.

By introducing a classification of design flaws, we aim to discover generic detection and correction methods and ease the assessment of new reengineering techniques and tools. Sorting and classifying

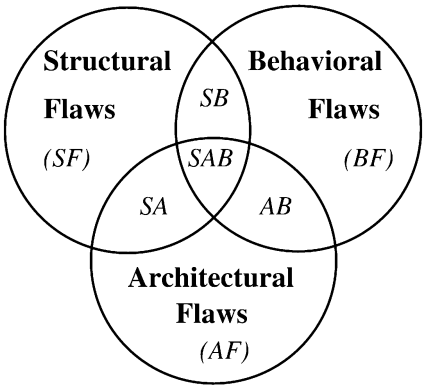


Figure 1. Design flaws classification.

design flaws is a complex task because of the multiple points of view that can be considered. We propose the following classification as it can be inferred from the related literature [5–10]. Such a design flaw classification can distinguish among: (i) design flaws involving the *internal structure* of a class; (ii) design flaws involving *interactions* among classes; and (iii) design flaws relating to the application *semantics*. We consider these three categories because they represent three distinct levels of abstraction and thus rely on different detection and correction techniques.

However, these three categories are not orthogonal and several design flaws do not fit simply into a single category. We can define four additional categories as depicted in Figure 1, which pertain to the intersections of the three major categories. A finer-grain classification of the design flaws is presented below.

- *Structural flaws (SF):* this category includes any design flaws related to the internal structure of a class. It embodies *stylish* and *syntactic* flaws, which are design defects in the structure of the class and its members. For example, methods with too many invocations are error-prone and difficult to maintain or extend [11].
- *Architectural flaws (AF):* this category encloses any design flaws related to the external structure of the classes (their public interface) and their relationships. All design flaws in the application architecture belong to this category. For example, mixing different algorithms within a single data structure is an architectural flaw. The reason for this is that the algorithms overweigh the data structure and any data structure extension is not easy because it requires modification every time a new algorithm is considered [12].
- *Behavioral flaws (BF):* this category encompasses all the design flaws related to the application semantics. For example, the 'The Year 2000 Problem' (due to the storage of years on only two digits) is a typical behavioral design flaw. Another example of behavioral design flaws concerns changes in the operating environment of a system.
- *Intersection of SF and AF (SA):* this category includes design flaws related to both the internal and external structures of the classes. There are some internal design flaws for which corrections

imply changes to the application architecture. For example, duplicated code among classes reveals a need to change the architecture to factor out the duplicated code. Also, there are some architectural design flaws involving changes to the internal structures of the classes. An example is the use of the *Composite Pattern* [12], where a specialized object may create a meaningful new object where we can attach domain specific behavior.

- *Intersection of SF and BF (SB):* this category pertains to design flaws involving both the semantics of the class and its internal structure. There are some defects in the behavior of the class which in order to be corrected require changing their structure and behavior.
- *Intersection of AF and BF (AB):* this category pertains to design flaws related to both architecture and behavior of the classes. There is a set of design flaws related to the application architecture which in order to be corrected requires changing the semantics of the classes involved. For example, a 'God' class [9] is a sign of a bad architecture that for its improvement requires changing the semantic of at least the 'God' class. Also, there are some design flaws in the behavior of classes which in order to be corrected need recursive changes in their architecture.
- *Intersection of SF, AF, and BF (SAB):* the last category includes the set of all the design flaws pertaining to the structure, semantics, and the architecture of the application.

Based on our proposed classification, it is possible to distinguish among design flaws relative to any combination of syntactic, structural, semantic, or architectural defects. These categories also allow differentiation among design flaws that stem from one category and imply changes in another. For example, duplicate code across classes is detected in internal structures (SF) of the classes, but resulting flaws appear in both internal structures (SF) and their architecture (AF). Moreover, the concern for improving the quality of the OO design of legacy systems is related to applying the changes which preserve the behavior of the system. Since we focus this research on SF, AF, and the intersections between them that can cause decreasing design quality, we exclude from this work pure behavioral flaws that may occur in an OO application.

3. QDR FRAMEWORK

It is widely accepted that the reengineering of legacy systems has become a major concern in today's software industry. Traditionally, most reengineering efforts were focused on systems written in traditional programming languages such as Fortran, COBOL, and C [13–17]. Unfortunately, none of such efforts embeds to the reengineering process quality requirements for the migrant system as a goal. In this context, we proposed a *quality-driven reengineering (QDR) framework* [2] which 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 source code level of the legacy system being reengineered.

To represent information about different software qualities, their interdependencies, and the software transformations that may affect them, we adopt the non-functional requirement (NFR) framework proposed in [18]. 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/software development. A *soft-goal interdependency graph* is used to support the systematic, goal-oriented process of associating design decisions with software qualities. According to

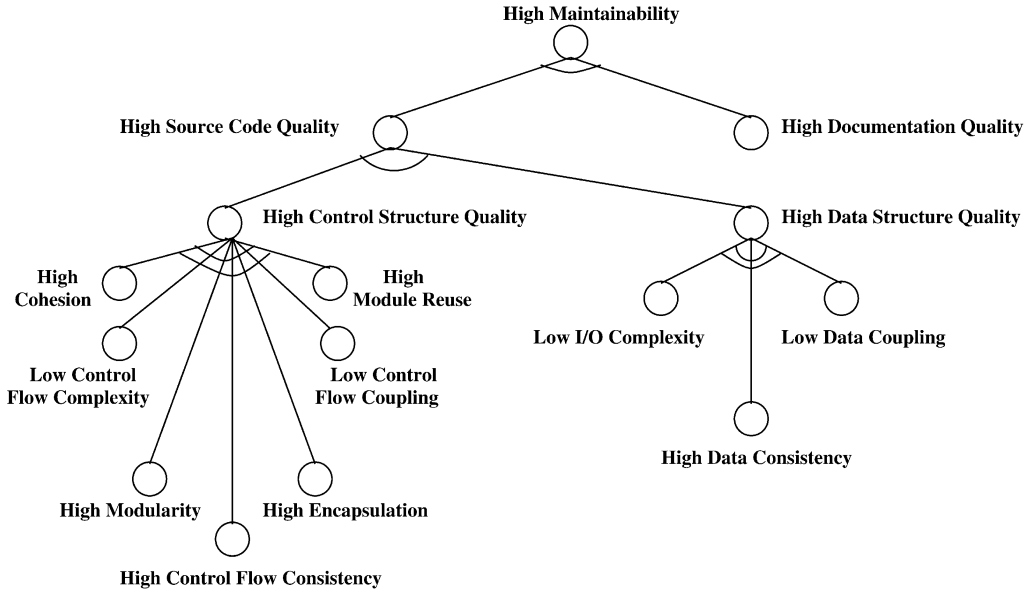


Figure 2. Maintainability soft-goal graph decomposition.

the framework, software qualities are represented as soft-goals, i.e., goals that can be partially achieved. The leaves of the soft-goal interdependency graph represent design decisions which fulfill or contribute positively/negatively to the soft-goals above them. Given a quality constraint for a reengineering 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 the additional design decisions or software transformations that may affect the desired quality positively or negatively.

The reengineering process in the QDR framework includes these steps. First, the source code is represented as an Abstract Syntax Tree (AST) [19]. The tree is further decorated using a linker, with annotations that provide linkage, scope, and type information. In a nutshell, once software artifacts have been understood, classified and stored during the reverse engineering phase, their behavior can be made readily available to the system during the forward engineering phase. Then, the forward engineering phase aims to produce a new version of an OO legacy system that operates on the target architecture and aims to address specific non-functional requirements (i.e., maintainability enhancements). Finally, we use an iterative procedure to obtain the new migrant source code by selecting and applying a transformation which leads to 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 target quality requirements are met.

Figure 2 shows a sample soft-goal interdependency graph for maintainability. This graph represents and models a set of software attributes that relate to software maintainability. The graph was compiled after a thorough review of the literature [19–21]. In Figure 2, *AND* relations are represented with a

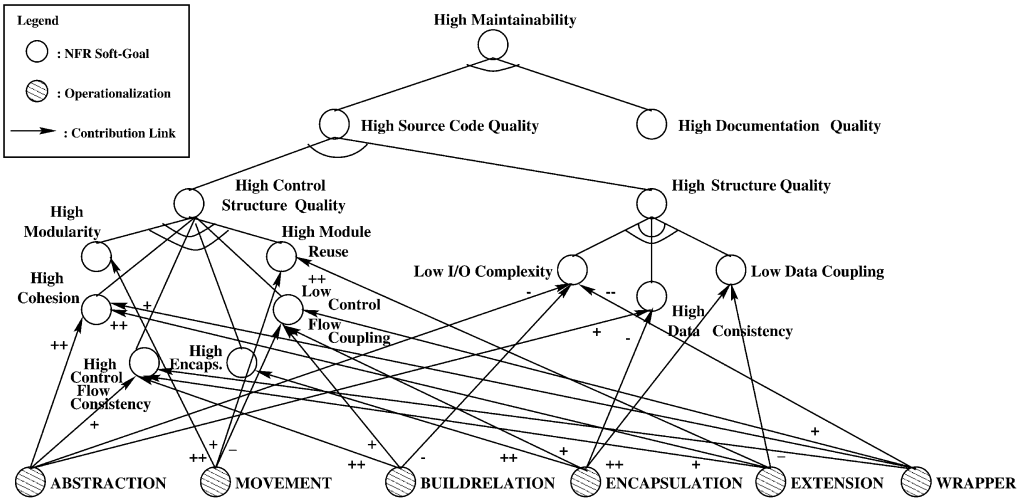


Figure 3. Maintainability soft-goal graph with meta-pattern transformations.

single arc, and *OR* relations with a double arc. While NFR soft-goal nodes as illustrated in Figure 2 provide specific interpretation of what the initial NFR of *Maintainability* is, they do not yet provide a means for guiding the transformation process and actually achieving the desired quality.

4. META-PATTERN TRANSFORMATIONS AS DESIGN MOTIFS

At some point during the reengineering process, when the non-functional requirements have been sufficiently refined, one must be able to identify, apply, and evaluate reengineering and software transformation actions for achieving these requirements (that are modeled as soft-goals). In this respect, we propose to associate meta-pattern transformations [4,23] with the soft-goal graphs as shown in Figure 3. The meta-pattern transformations illustrated in Figure 3 for maintainability are the design motifs that occur frequently; in this way it is similar to design patterns, but these are lower-level constructs [24]. We call these associations *operationalizations* of the NFR soft-goals [18]. Operationalizing soft-goals are drawn as shaded circles and are just another type of soft-goal graph node.

The proposed meta-pattern transformations [23,4] provide guidelines to operationalize a soft-goal dependency graph (i.e., for maintainability) as shown in Figure 3. While interested readers can refer to [23,4] for each transformation which comprises a precondition, an algorithmic description, a postcondition, and its impact on quality, Table I provides a brief summary of these meta-pattern transformations.

In this context, satisfying soft-goals yields a positive or negative contribution towards parent soft-goals in terms of *AND*, *OR*, $+$, $++$, or $-$, $--$ relations. Our aim is to assist the developer in improving

Table I. Meta-pattern transformations as design motifs.

Meta-pattern transformation	Description
ABSTRACTION (ABS)	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.
EXTENSION (EXT)	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 the 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.
MOVEMENT (MOV)	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.
ENCAPSULATION (ENC)	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.
BUILDRELATION (BRL)	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.
WRAPPER (WRP)	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.

the quality of a system during reengineering by providing guidance as to which transformations are the best to apply. 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 (Figure 3). One possible alternative is to use the ABSTRACTION meta-pattern transformation as shown in Figure 3. 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. In this respect, we say that the ABSTRACTION transformation *operationalizes* high cohesion. We also say that the high cohesion NFR is *operationalized by* the ABSTRACTION transformation.

5. ROLE OF DESIGN FLAWS IN THE QDR FRAMEWORK

It is widely accepted that the design of a large system deteriorates with each evolution cycle. Consequently, we aim to devise a framework whereby such design flaws introduced during software evolution could be identified and corrected.

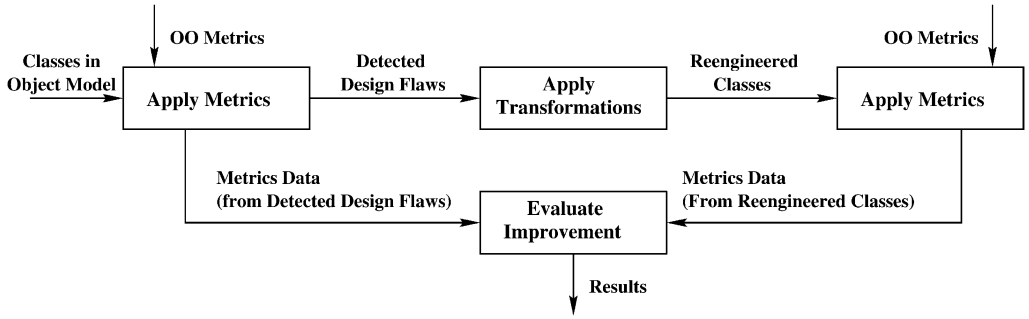


Figure 4. Reengineering strategy for design flaws.

Figure 4 illustrates the proposed reengineering strategy using OO metrics in order to evaluate and assess the impact that transformations have on soft-goals. As is known, an object model has several levels of representation, including *application level*, *subsystem level*, *class level*, and *function level* [25]. While design flaws can occur at any level, our focus here is on class level deterioration. Improving deteriorated classes is one major step to keeping OO legacy systems operational. After extracting an object model at the class level through reverse engineering, the proposed strategy is depicted in Figure 4 and employs the following steps.

- *Step 1:* to select, measure, and record the specific OO metrics in order to detect classes for which quality has deteriorated. While there are several reasons that a design may lose quality over time, here the focus is on detecting the classes that have high complexity and high coupling. For detecting such classes, there is a need to have a classification of OO metrics which relate to different categories of design quality and source code features. In Section 6, we propose and discuss a useful catalogue of such metrics.
- *Step 2:* to reengineer detected design flaws using proper transformations. For correcting such design flaws through software transformations, we need to study the impact that specific transformations have on specific metrics. In Section 7, we discuss the impact of the proposed software transformation framework on the selected OO metrics. Based on the preconditions for each transformation and the source code features, we identify all possible transformations that can be applied at any given point of the transformation process.
- *Step 3:* to re-apply and record the same object-oriented metrics to the reengineered classes and finally compare the recorded results to evaluate design and source code improvement as well as compliance with the desired requirements. Once the transformation is determined and applied, it is necessary to verify that a transformation contributes towards the desired target qualities.

One way to detect design flaws at the class level is to identify violations of a 'good' OO software design by performing source code analysis. Even though there is no consensus of what constitutes a good design, some general guidelines and principles have been proposed in the literature [9]. While there are several reasons that a class may lose quality over time, here the focus is on the

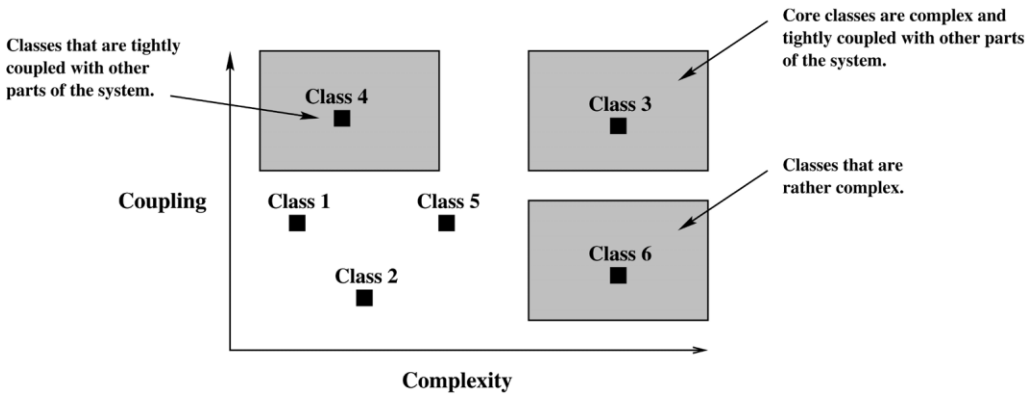


Figure 5. Key classes in an OO legacy system.

classes that have high coupling and low cohesion. These characteristics often result in the loss of abstraction and encapsulation. In particular, they are those highly-coupled classes that often loose cohesion during the course of development. Based on this assumption, two fundamental quality design heuristics are proposed to detect design flaws at the class level and are elaborated further in the following sections.

5.1. Key classes heuristic (KCH)

A proper way to detect design flaws at the class level is to identify which classes implement the key concepts of the system. Usually, the most important concepts of a system are implemented by very few *key classes* [26] which can be characterized by the specific properties. These classes, which we refer to as *key classes*, manage many other classes or use them in order to implement their functionality. The key classes are tightly *coupled* with other parts of the system. Additionally, they tend to be rather *complex*, since they implement much of the legacy system's functionality.

Identifying these classes is a starting point in the proposed framework to detect potential design flaws and to correct them properly based on the proposed software transformation framework. Figure 5 illustrates such an analysis. The classes of the OO legacy system are placed into a coordinate system according to their complexity and coupling measurements. Classes that are complex and tightly coupled with the rest of the system fall into the upper-right corner and are good candidates for these *key classes*. Mathematically, we can combine two or more metrics by computing the *distance d* of a class from the origin of the coordinate system. If $\bar{x}$ denotes the complexity metrics vector value of a class c, and $\bar{y}$ its coupling metrics vector value, we compute the combined value d_c as:

$$d_c(\bar{x}, \bar{y}) = \max(d_c(x, y)), \quad \forall x \in \bar{x},\ y \in \bar{y} \tag{1}$$

where $d_c(x, y) = \sqrt{x^2 + y^2}$. In some cases, if the metrics use a very different scale, some normalization might be required and we can then use the following formula instead:

$$d_c(x, y) = \sqrt{\left(\frac{x}{x_{\max}}\right)^2 + \left(\frac{y}{y_{\max}}\right)^2} \tag{2}$$

This combined value allows for class comparison. Classes with higher values for d are better candidates to be considered as key classes of the system than classes with lower values for d. The value of d provides a good means to identify the key classes of the system that may represent design flaws which need to be taken care of.

5.2. One class–one concept heuristic (OC2H)

A very basic principle in object-oriented software engineering is that a class should implement one single concept of the application domain. Some violations of this principle can be detected by using the assumptions: (i) that a class that implements more than one concept probably has low cohesion measurements, since these concepts can be implemented separately; and (ii) that a class that by itself does not implement one concept (the implementation of the concept is distributed among many classes) is probably tightly coupled to other classes.

Therefore, by collecting cohesion and coupling values of an OO legacy system, possible violations of the principle 'one class–one concept' can be found. These classes tend to have either low cohesion values or high coupling values. The classes that have very low cohesion values can often be split [27]. Sometimes this leads to a more flexible design, since the two separate classes are easier to understand and are more reusable. Low cohesion values also indicate deteriorated classes. These classes are not implementing a self-contained object from the application domain, they just group methods together, acting as a module.

6. OO METRICS SUITE

Each of the design flaws identified in Section 2 and each of the quality rules for detecting these flaws represent an attribute or characteristic of a design. These characteristics are sufficiently well defined to be objectively assessed by using one or more OO metrics. Metrics can be particularly suitable to assess whether the OO legacy system adheres to design principles or contains violations of these principles.

In this section, we select OO metrics that will be used to assess OO system qualities in our quality-driven reengineering framework. In order to make a selection, we first need to establish a set of criteria that should guide the selection process. Establishing these criteria requires consideration and identification of which of the metrics can be successfully used in order to assess the improvement of the quality of the migrant system and to collect proper transformations based on the source code features. In this respect, we focus on two criteria: (i) the theoretical evaluation of the definition of the metric; and (ii) the aspects of design flaws that we plan to detect and correct.

The proposed selection of the OO metrics is classified according to four major metrics categories [28]: complexity metrics, coupling metrics, cohesion metrics, and inheritance metrics.

While Table II illustrates these metrics along with their relation to soft-goal nodes, we further describe them in a more detail as follows.

- *Complexity metrics:* we consider these metrics because they may provide indications about the level of complexity for a given class. One of the well-established metrics to measure the complexity of a class is *WMC* (weighted methods per class) which measures the complexity of a class by adding up the complexities of the methods defined in that class [29,30]. A special case of *WMC* (which is very simple to compute) is *NOM* (number of methods) as well as the *RFC* (response set for a class) metric [29], which measures the complexity of a class by counting the number of methods defined in that class [31]. Such metrics measure the attributes of the objects in the class and express the potential communication between the class that is measured with other classes, i.e., how many methods local to the class and methods from other classes can be potentially invoked by invoking methods from the class. Complexity measurements for methods are usually given by code complexity metrics like lack of cohesion (*LOC*) or the McCabe Cyclomatic complexity [32]. The class definition entropy (*CDE*) [33] metric identifies complex classes in an OO system. Classes with higher values of *CDE* can be expected to have a complex implementation and a more-than-average number of errors and changes. Obviously, complexity metrics play an important role when reengineering software systems, as classes with high complexity measurements are difficult to understand and consequently difficult to change.
- *Coupling metrics:* another important aspect when dealing with an OO legacy system is the coupling level between classes. A class is coupled to another class, if it depends on that class, for example by accessing the variables of that class, or by invoking methods from that class. Classes that are tightly coupled cannot be seen as isolated parts of the system. Understanding or modifying them requires that other parts of the system must be inspected as well. Conversely, if other parts of a system change, classes with high coupling measurements are more likely to be affected by these changes. Additionally, classes with high coupling tend to play key roles in the system, making them an appropriate starting point when trying to understand an unfamiliar object-oriented legacy system. Analyzing the viewpoints suggested for the different coupling metrics, one is able to reason on the level of reuse and maintainability of a class. Specifically, *DAC* (data abstract coupling) measures coupling between classes that results from attribute declarations [33–35]. *DAC* counts the number of abstract data types defined in a class. Essentially, a class is an abstract data type, therefore *DAC* reflects the number of declarations of complex attributes (i.e., attributes that have another class of the system as a type). *CDBC* (change dependency between classes) is another interesting metric that directly addresses an important aspect in reengineering; that is, maintenance effort.
- *Cohesion metrics:* the cohesion of a class describes how closely the entities of a class (such as attributes and methods) are related. Often, cohesion is measured by establishing relationships between methods of the class in the case where the same instance variables are accessed. A useful metric measuring this property is *TCC* (tight class cohesion) [34,35,37,38], which measures the cohesion of a class as the relative number of directly connected methods, where methods are considered to be connected when they use at least one common instance variable. In the literature, several formulas have been introduced to compute LOC [29,31,36]. For this paper, we

adopt the definition in [31] which measures dissimilarity among all the methods of a class except the inherited methods, but including overloaded methods. The *LCOM* value denotes the number of pairs of methods without shared instance variables, minus the number of pairs which do share instance variables.

- *Inheritance metrics:* this category of metrics attempts to provide indicators on the quality of the class hierarchy of an OO legacy system. A useful metric measuring this property is *DIT* (depth of inheritance) [29]. Basili *et al*. [39] argued in their report that the information that they obtained from this metric was useful in reasoning about the quality of the class. The number of children (NOC) [36] represents the number of immediate subclasses subordinated to a class in the class hierarchy. The greater the number of children, the greater the reuse, since inheritance is a form of reuse. An immediate conclusion is that the classes with a greater number of children have to provide more services in different contexts, and thus they should be more flexible. An assumption that can be made is that such classes will introduce more complexity in the design.

The objective of associating metrics and soft-goals as illustrated in Table II is to identify a small set of metrics which contain sufficient information to allow an evaluation of source code features and changes pertaining to design properties during the application of software transformations.

7. IMPACT OF APPLYING META-PATTERN TRANSFORMATIONS ON METRICS

Once a source code fragment is selected as a candidate for reengineering using the OO metrics presented above, the next step is to propose possible transformations that improve the quality of the program while preserving its behavior. The suggested thresholds of what range of values contributes a good or bad design with respect to each metric are based on proposed ranges of values presented in [39,40]. In this respect, we establish a cause-to-effect relationship between some combinations of metrics and a poor design quality. Therefore, the problem to address is what transformations and code changes should be applied to improve the corresponding metrics and, therefore, the corresponding system quality. An intuitive solution is to identify which transformation (or a set of transformations) positively affects the value of a particular metric (or a set of metrics) that releases to the specific qualities being improved. To respond to such a question, we need to consider two steps: (i) propose a catalogue of transformations as a predefined set of transformations that can be applied both at the internal and external structures of the classes; and (ii) analyze the impact of each transformation on the predefined set of metrics.

In this context, there is a synergy between design heuristics and design patterns. Design heuristics can highlight a problem in one facet of a design, while patterns can provide the solutions. In this work, the proposed transformations alter the design with the purpose of improving a specific quality of the system while preserving its behavior. These transformations modify the structure of a program, which possibly modifies the values of the metrics related with the quality being improved in a positive way. As we are interested in class-level metrics, we study the metric variations for all classes involved in a transformation. The possible impact of applying each transformation on metrics for the classes involved is shown in Table III. Note that '+' means that there is a positive impact, '−' means that there is a negative impact, and '*NI*' means that there is no impact.

Table II. Selected object-oriented metrics and their relationships with soft-goals.

Metric name	Category	Definition	Soft-goal nodes
CDBC [35]	Coupling	*CDBC* determines the potential amount of follow-up work to be done in a client class (*CC*) when the server class (*SC*) is being modified defined as $CDBC(CC, SC) = \min(n, A)$ where $A = \sum_{\text{implement}=1}^{m_1} \alpha_i + (1 - k) \sum_{\text{interface}=1}^{m_2} \alpha_i$.	Low control flow coupling, high encapsulation, high control flow consistency
CDE [33]	Complexity	*CDE* computes a decimal number to indicate a class definition complexity based on the usage and frequency of different name strings in a class and defined as $CDE = -\sum_{i=1}^{n_1} (f_i/N_1) \log(f_i/N_1)$.	Low control flow complexity, high cohesion, high encapsulation
DAC [36]	Coupling	*DAC* measures the coupling complexity caused by ADTs.	Low data coupling, high data consistency, high encapsulation
DIT [36]	Inheritance and Coupling	*DIT* represents the length of the tree from that class to the root of the inheritance tree.	High module reuse, high encapsulation, high modularity
LCOM [31]	Cohesion	Consider a class C, its set M of m methods $M_1, \ldots, M_m$, and its set A of a data members $A_1, \ldots, A_a$ accessed by M. Let $\mu(A_k)$ be the number of methods that access data attribute A_k where $1 \leq k \leq a$. Then, $LCOM(C(M, A)) = [((1/a) \sum_{j=1}^{a} \mu(A_j)) - m]/1 - m$.	High cohesion, high data consistency, low I/O complexity, high modularity, high module reuse
LD [35]	Coupling	*LD* is determined by relating the amount of data local to the class to the total amount of data used by the that class and is defined as: $LD = (\sum_{i=1}^{n} \lvert L_i \rvert)/(\sum_{i=1}^{n} \lvert T_i \rvert)$.	Low data coupling, high data consistency, high encapsulation
NOC [36]	Inheritance and Coupling	*NOC* represents the number of immediate subclasses subordinated to a class in the class hierarchy.	High module reuse, high encapsulation, high modularity
NOM [36]	Complexity and Coupling	Since the local methods in a class constitute the interface increment of the class, *NOM* serves the best as an interface metric and is defined as the *number of local methods in a class*.	Low I/O complexity, high cohesion, high modularity
RFC [29]	Complexity and Coupling	The response set for a class (*RS*) is a set of methods that can be potentially executed in response to a message received by an object of that class and is defined as $RFC = \lvert RS \rvert$.	Low control flow coupling, low control flow complexity, low I/O complexity, high control flow consistency
TCC [37]	Cohesion	Let $NP(C)$ to be the total number of pairs of abstract methods in $AC(C)$ and $NDC(C)$ to be the number of directed connection in $AC(C)$, then *TCC* is defined as: $TCC(C) = NDC(C)/NP(C)$.	High cohesion, high modularity, high module reuse
WMC [29]	Complexity	Consider a class C_1 with methods $M_1, \ldots, M_n$, and $c_1, \ldots, c_n$ are the static complexity of the methods, then $WMC = \sum_{i=1}^{n} c_i$.	Low control flow complexity, high control flow consistency, low I/O complexity

Table III. Impact of the transformations on the OO metrics suite.

	Metric name										
Transformation	CDBC	CDE	DAC	DIT	LCOM	LD	NOC	NOM	RFC	TCC	WMC
ABSTRACTION	+	+	+	*NI*	+	+	*NI*	+	+	+	−
EXTENSION	+	+	−	+	+	−	+	+	*NI*	+	*NI*
MOVEMENT	−	*NI*	*NI*	+	+	*NI*	+	+	−	+	*NI*
ENCAPSULATION	+	+	+	+	−	+	+	*NI*	*NI*	*NI*	*NI*
BUILDRELATION	+	+	*NI*	*NI*	−	*NI*	*NI*	−	+	*NI*	+
WRAPPER	+	+	*NI*	*NI*	−	*NI*	*NI*	−	*NI*	+	−

8. A MODEL FOR A SIG REPRESENTATION

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

As illustrated in Figure 3 for the decomposition of SIGs, the development of the graph proceeds by repeatedly refining *parent* soft-goals into *offspring* soft-goals. In such refinements, the offspring can contribute fully or partially, and positively or negatively, towards satisfying the goals denoted by the parent goal node. The contribution operators are *AND*, *OR*, +, ++, −, −−. For our purposes, the arcs of the directed graph for each SIG are labeled by these operators. A SIG is given by the *start node* called s, representing the top-level requirement state, and the above rules associate goals (parent nodes) and sub-goals (children nodes). It is convenient to model the above rules in terms of contribution operators.

The first two contribution operators, *AND* and *OR*, relate a group of offspring to a parent. To keep track of the information regarding these two contribution operators, we build an adjacency matrix, namely the *soft-goal adjacency matrix* (*SAM*). Let n_1 and n_2 be vertices of a SIG. If n_1 and n_2 are joined by an arc e with either an *AND* or *OR* contribution operator, then n_1 and n_2 are said to be *adjacent* with the defined rule. If the arc e is directed from n_1 to n_2, then the arc e is said to be *incident from* n_1 and *incident to* n_2. Let D to be a SIG in digraph notation, with n vertices or soft-goal nodes, $N = \{d_1, d_2, \ldots, d_n\}$. The simplest graph representation model uses an $n \times n$ matrix *SAM* of &'s, |'s, and 0's given by

$$SAM_{i,j} = \begin{cases} \& & \text{when } AND(d_i, \{d_j\}) \\ | & \text{when } OR(d_i, \{d_j\}) \\ 0 & \text{otherwise} \end{cases}$$

that is, the (i, j)th element of the matrix is not equal to 0 only if $d_i \longrightarrow d_j$ is an edge in D with a contribution operator.

As mentioned before, there are two types of nodes in a SIG, namely *NFR soft-goal* nodes and *transformation operationalization* nodes. In practice, a transformation node is considered for the purpose of implementing a subset of the NFR soft-goals. Thus, each NFR soft-goal is associated with a set of transformations according to what features of the source code a transformation affects, and according to the soft-goal nodes involved. The NFR soft-goal allocation is not necessarily one-to-one—that is, a single soft-goal may be associated to more than one transformation.

More formally, let n_1 to be a soft-goal node and n_2 a transformation or operationalization node. If n_1 and n_2 are joined by an arc e with any of the $++$, $+$, $-$ or $--$ contribution operators, then n_1 and n_2 are said to be *adjacent* with the defined rule. The arc e which is directed from n_2 to n_1 is said to be *incident from* n_2 and *incident to* n_1. Let S to be a set nodes, $S = \{s_1, s_2, \ldots, s_n\}$, representing soft-goals and T is a of set operationalization nodes, $T = \{t_1, t_2, \ldots, t_m\}$, representing transformations. The simplest impact representation scheme uses an $m \times n$ matrix *TIM* (transformation impact matrix) of $++$'s, $+$'s, $--$'s, $-$'s and 0's given by:

$$TIM_{i,j} = \begin{cases} ++ & \text{when } (t_i, s_j, ++) \\ + & \text{when } (t_i, s_j, +) \\ - & \text{when } (t_i, s_j, -) \\ -- & \text{when } (t_i, s_j, --) \\ 0 & \text{otherwise} \end{cases}$$

that is, the (i, j)th element of the matrix is not equal to 0 only if $t_i \longrightarrow s_j$ is an edge in D with a contribution operator.

As discussed above, the building process of a goal graph can be divided into two steps. First, the initial goal will be split into sub-goals. Rules based on contribution operators can be applied to each of these sub-soft-goals independently. The result of these applications can also be split and so on until there is no further splitting possible based on the existing information. This step creates a *SAM*. Second, we can generate a *TIM* based on soft-goals, transformations and their relationships.

9. A TRANSFORMATION SELECTION ALGORITHM

Based on the discussion in Section 5, the algorithm shown in Figure 6 summarizes the detection and correction activities as implemented by our proposed reengineering framework.

Consider, *AC* to be a set of classes in an object model extracted from an OO legacy system. We need to calculate the metrics values from the predefined catalogue and apply quality heuristics rules to detect design flaws and *deteriorated classes* in the legacy system being analyzed.

In this process, the first step is to apply the *key classes (KCH)* rule (line 4 in Figure 6) by using both complexity and coupling metrics. A very-high-level quality goal for a software system could be maintainability, thus coupling measurements should not be high in order to ensure that changes to the system do not trigger changes throughout the system. Here, for the purpose of simplicity, we consider only *DAC* as a coupling related metric. Therefore, monitoring *DAC* values can be promising. When a

Selection Algorithm. $SGE(\mathcal{AC}, \mathcal{SAM}, \mathcal{TIM}, \mathcal{MC}) =$
Input:
$\mathcal{AC}$: A set of classes in an object-oriented legacy system.
$\mathcal{SAM}$: Soft-Goal Adjacency Matrix.
$\mathcal{TIM}$: Transformation Impact Matrix.
$\mathcal{MC}$: Object-Oriented Metrics Suite.
Output:
$\mathcal{AC}$: Modified the set of classes after applying meta-pattern transformations.

Variables:
$\mathcal{COH}_i$: Cohesion Metrics Vector for class i.
$\mathcal{COU}_i$: Coupling Metrics Vector for class i.
Method:

```
1. for each class C in AC do begin
2.     COH_C = NIL;   COU_C = NIL;
3.     calculate the metrics values from MC;
       /* Detect design flaws based on metric value by applying the evaluating rules */
4.     append(COU_C, DAC);          /* apply KCH Rule */
5.     append(COH_C, RFC);          /* apply OC2H Rule */
6.     ....     /* Other coupling/cohesion evaluation rules using metrics from MC can also be applied here */
7.     if C is a deteriorated class based on d_C(COU_C, COH_C) then begin
8.     select a set of potential transformations {T_i} that can correct design flaws using TIM and SAM;
9.     apply the transformations T_j ⊂ T_i that corresponds to the context of C because of its features;
10.    end-if
11. end-for
```

Figure 6. Description of the transformation selection algorithm.

significant number of classes evolves to higher *DAC* measurements, some *refactoring operations* [11] or meta-pattern transformations [23,4] of the system could be appropriate for reducing coupling.

Moreover, good OO design styles usually require that classes have high cohesion, since they should encapsulate concepts that belong together. Classes with low cohesion often represent violations to a flexible, extensible or a reusable design. All of these are issues that must be dealt with during the OO reengineering process. Therefore, by applying cohesion metrics like *TCC* and coupling metrics like *DAC* and *RFC* to the OO legacy system, possible violations of the principle OC2H rule (line 5 in Figure 6) can be found. These classes tend to have either low *TCC* values or high *DAC* and *RFC* values.

For example, classes that have very low *TCC* values, can often be split [11]. Sometimes this leads to a more flexible design, since the two separate classes are easier to understand and are more reusable. Low *TCC* measurements may indicate classes that have not been designed in an OO way. These classes are not implementing a self-contained object from the application domain, they just group methods together, acting as a module. If a class exhibits low method cohesion, it indicates that the design of the class has probably been partitioned incorrectly. In this case, the design could be improved if the class is split into more classes with individual higher cohesion values. The *LCOM* metrics help to identify and assess such design flaws.

10. A PROTOTYPE FOR THE QDR FRAMEWORK

As part of this work, the proposed QDR approach has been implemented in a prototype which offers an interactive and incremental environment for improving the quality of an OO system. The current version of the QDR framework prototype consists of 6000 lines of code written in Java. The architectural design of this prototype consists of a number of components with simple interfaces and with a *pipe and filter* architectural style. Each component (filter) processes its input data in the form of a file (pipe) and stores the results in another file for the next component. Figure 7 illustrates the architecture of this prototype with its components which are elaborated further.

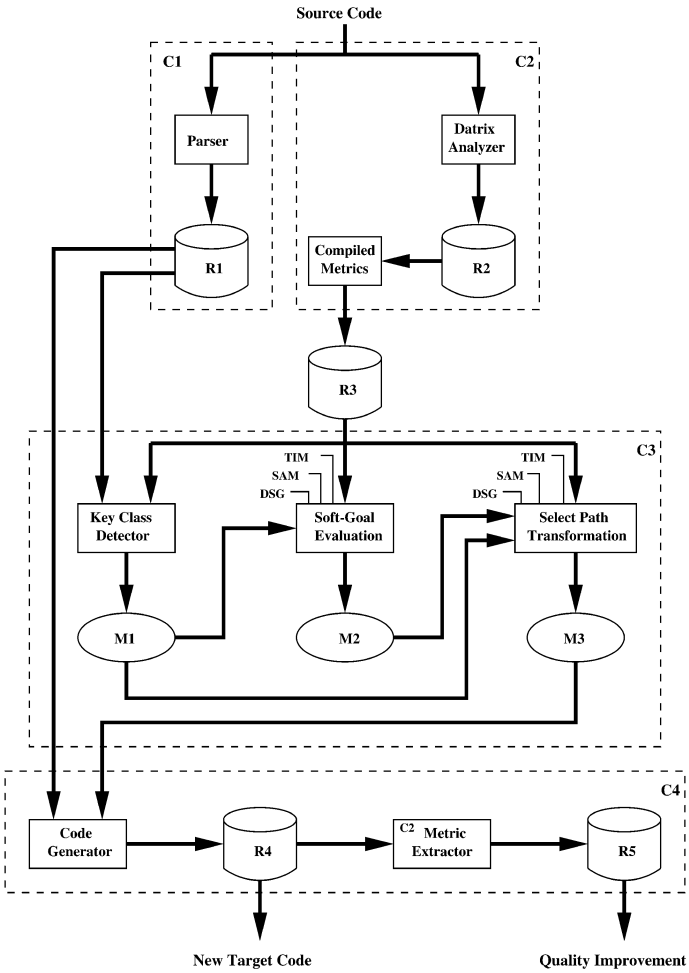
10.1. Pre-process components

These components are composed of two phases which are described as follows.

- *Parsing phase:* program representation plays an important role in building tools that facilitate software analysis and software maintenance. One of the most popular representations is the AST [19]. We have used Ret4J [41] for building ASTs as a program representation scheme. Ret4J defines a language model in terms of a Java language DTD and automatically annotates source code with XML tags. The benefit of using XML in this context is that the corresponding tree conforms with the language domain model as defined in the corresponding language DTD. The advantage of this method is that the complete document exists in memory and can be easily processed and manipulated. The disadvantage is that working with large XML documents imposes a large memory requirement. This phase has been depicted (C1) on the left-hand side of Figure 7.
- *Metric extractor phase:* metrics are not indications of bugs, but high metric numbers often indicate how complex, fragile or sensitive to regressions a class may be. Classes or methods with high metric values may be good candidates for refactoring and software transformations. It is therefore useful for us to monitor metrics and investigate those metrics which seem out of line. *Datrix* [42], which is a tool for software evaluation and a trademark of Bell Canada, helps us in this respect. This phase is done in two stages. First, direct class metrics (37 in total) are extracted from the source code using Datrix Metric Analyzer. Second, the selected indirect metrics which were discussed in Section 6 can be computed based on the first step running some written scripts, as a part of the prototype, which is the most attractive feature of Shell Programming. This phase, including the two steps, has been depicted (C2), on the right-hand side of Figure 7.

10.2. Analysis components

The major tasks of the prototype are performed by three analysis components as shown in the middle part of Figure 7 (C3). After parsing the source code and extracting the selected metrics, it is time to improve the quality of the parsed OO system. An object model has several levels of representation, including *application level*, *subsystem level*, *class level*, and *function level* [25]. While design flaws can occur at any level, our focus here is on class level deterioration. While there are several reasons for a design to lose quality over time, here the focus is on detecting the key classes, using *Key Class Detector*, that have high complexity and high coupling, and low cohesion. On the other hand, the reengineering



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

Figure 7. The architectural design of the prototype for the QDR framework.

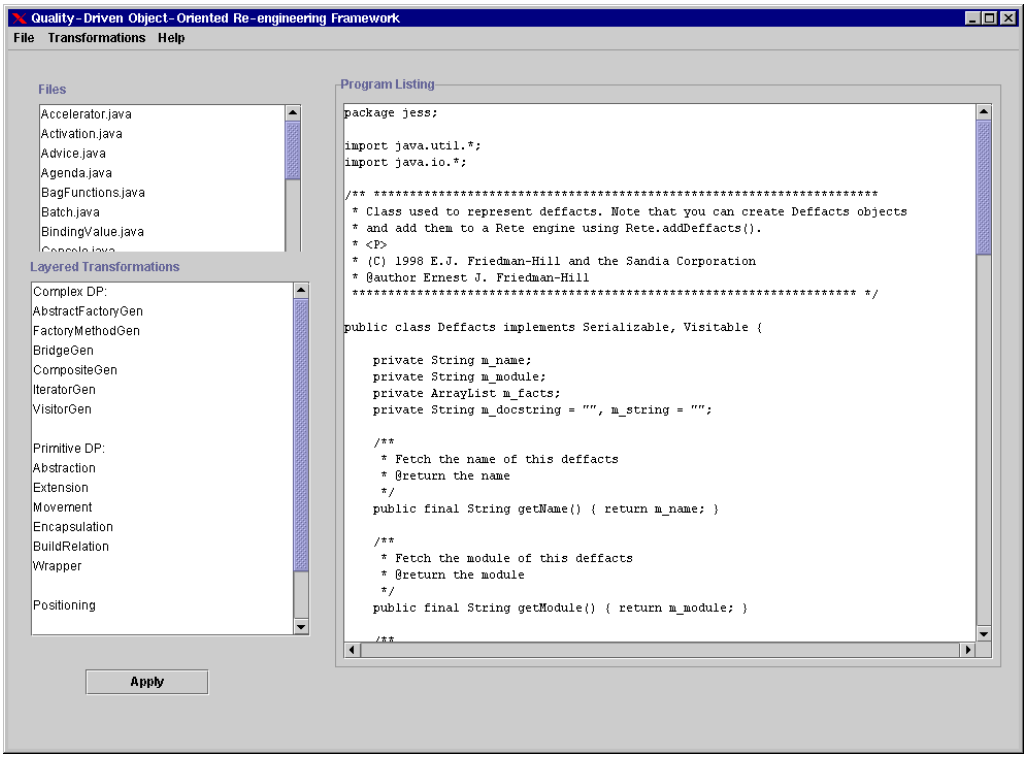


Figure 8. A snapshot of the QDR framework.

activities need to be performed on specific non-functional requirements that are represented as a list of desired soft-goals (DSGs). The *Soft-Goal Evaluation* component builds a search graph of potential transformations based on: (i) *SAM*, which was discussed in Section 8; (ii) *TIM*, which was discussed in Section 8; and (iii) the list of DSGs. The evaluation algorithm was elaborated on in Section 9. The final step in this phase (C3) is responsible for identifying the set of all solution graphs for selecting source-code improving meta-pattern transformations. Based on the preconditions for each transformation and the source code features, we identify all possible transformations that can be applied at any given point of the transformation process. A snapshot of the QDR framework is depicted in Figure 8.

10.3. Post-process components

Once the meta-pattern transformations are determined and applied, it is necessary to verify that they contribute towards the desired target qualities. The last step of the implemented prototype as shown in Figure 7 (C4), takes care of these requirements. The *code Generator* component uses XML2Java from Ret4J [41] to convert back XML representation of an OO metric to the reengineered classes and finally

Table IV. Source code statistics of the case study software systems.

System name	Source code (lines)	Number of files	Number of classes
NetBeans JavaDoc	14 400	101	98
JESS	21 600	112	72
Eclipse Ant	34 800	178	100
Eclipse JDTCore	147 600	741	258

compare the recorded results to evaluate design and source code improvement as well as compliance with the desired requirements which can be computed using the described *metric extractor component*.

11. CASE STUDIES

To illustrate the approach proposed in this paper, we present a set of cases studies. The experiments are performed on four open-source software systems. Table IV presents the source code related characteristics of the experimentation suite. The experiments for each system involve the following.

(1) Java Expert System Shell [43], which is a rule engine and scripting environment written entirely in Sun's Java language by Ernest Friedman-Hill at Sandia National Laboratories. JESS was originally inspired by the CLIPS expert system shell, but has grown into a complete, distinct, dynamic environment of its own. Using JESS, one can build Java applets and applications that have the capacity to pertain inferences using a knowledge base in the form of declarative rules and facts. The core JESS language is still compatible with CLIPS, in that many Jess scripts are valid CLIPS scripts and vice-versa. Like CLIPS, JESS uses the Rete algorithm [44] to process rules, a very efficient mechanism for solving the difficult many-to-many matching problem.
(2) NetBeans JavaDoc [45], which supports the JavaDoc standard for Java documentation—both viewing it and generating it. JavaDoc is the Java programming language's tool for generating API documentation. The NetBeans IDE [46] is a development environment—a tool for programmers to write, compile, debug and deploy programs. It is written in Java, but can support any programming language. It is a free product with no restrictions on how it can be used. NetBeans JavaDoc gives the developers a solid documentation tool when working with code that the IDE lets us integrate API documentation for the code we are working on into the IDE itself. Jar files of classes imported by NetBeans JavaDoc are tools.jar and netbeans-support.jar.
(3) The Eclipse Ant Package [47], which provides support for running the Apache Ant [48] build tool in the platform. Apache Ant is a Java-based build tool. In theory, it is like Make, but without Make's wrinkles. Ant is different. Instead of a model that it is extended with shell-based commands, Ant is extended using Java classes. Instead of writing shell commands, the configuration files are XML-based, calling out a target tree where various tasks get executed. Each task is run by an object that implements a particular task interface. The Eclipse Ant Package defines a number of task and data types and various infrastructure pieces which make Ant easier

Table V. Some time and space statistics of the case study software systems.

System name	Source code (lines)	AST XML (MB)	Parsing time (min:sec)	Stored metrics (KB)	Extracting metrics time (s)
NetBeans JavaDoc	14 400	35.9	4:45	619.7	72
JESS	21 600	43.1	5:25	739.3	98
Eclipse Ant	34 800	58.4	8:37	765.8	173
Eclipse JDTCore	147 600	402.2	40:14	2528.4	258

and more powerful. Jar files of classes imported by Eclipse Ant are j2ee.jar, jakarta-org-2.0.5.jar and log4j-core.jar.

(4) The Eclipse JDTCore Package [49], where the Java model is the set of classes that model the objects associated with creating, editing and building a Java program. This package contains the Java model classes which implement Java-specific behavior for source code and further decompose Java resources into model elements. Jar files of classes imported by Eclipse JDTCore are ant.jar, jakarta-ant-1.4.1-optional.jar, resources.jar, runtime.jar and xerces.jar.

As discussed in Section 10, for detecting design flaws, and performing proper transformations of any kind, the Java source code and/or the Java class file must be parsed. The Re-engineering Tool Kit for Java parses the source code (JavaML) and generates XML documents that are easier to read and work with. Also, for collecting software metrics, we use the Datrix tool [42]. Our experiments were carried on a SUN Ultra 10 440 Mhz, 256 MB memory, 512 swap disk.

11.1. Time and space complexity

In this section, we evaluate the time and space complexity of quality-driven reengineering technique as a function of source-code size. In Table V, we have summarized the results collected from Java source codes in the pre-process phase. We can see that the output file which is based on the JavaML representation [41] is much larger than the input file which is a plain Java source file. A closer examination of the Java grammar used [50] provides a good explanation of this. In the DTD for JavaML [50,41], we observe that expressions are nested in such a way, that every kind of expression is described in terms of other expressions. This means that to represent a unary expression, there is a need to store many levels of other expressions in the XML file.

In terms of time requirements, in Figure 9 we also show the relationship between system size, AST XML size, and parsing time. Overall, the parsing of Java source files is reasonable, even when large systems are considered as seen in the last row in Table V. In the last two columns of Table V, we have summarized the results collected for metrics in terms of time and space. First, direct class metrics (37 in total) are extracted from the source code using the Datrix Metric Analyzer [42]. Second, the selected indirected metrics which were discussed in Section 6 can be computed based on the first step running some written scripts.

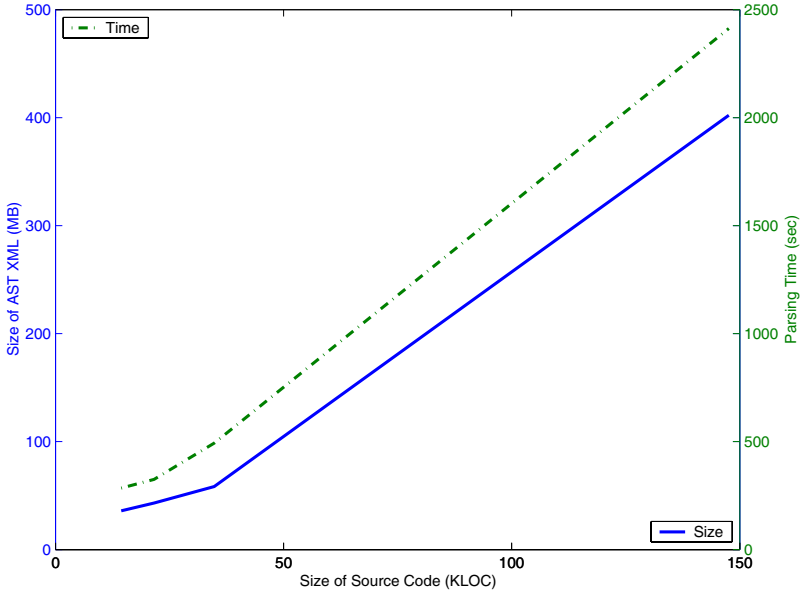


Figure 9. Graph of parsing time and size results for the case studies.

From Table V, we can see that the space for metrics is based on the input files which are plain Java source files. In terms of time requirements, in Figure 10 we also show the relationship between system size, size of stored metrics, and time for extracting the metrics. Overall, the time for extracting the metrics from Java source files is very fast, even when very large systems are considered as seen in the last row in Table V.

11.2. Impact of applied transformations on soft-goals

In this section, we summarize the collected results from case studies after applying the transformations and discuss their impacts on soft-goal nodes. In Table VI, we have summarized the results collected from case studies after applying the transformations to the OO metrics suite which are proper indicators for maintainability.

Good OO design recommends that classes be as lightly coupled as possible, so high *CDBC* [35], *DAC* [36], and *LD* [35] metric values may indicate poor design. Experimental results for the 'ABSTRACTIONS' transformation indicates an improvement for low coupling by its positive impacts on *CDBC* [35] at the level of 12% on average, on *DAC* [36] at the level of 10% on average, and on *LD* [35] at the level of 18% on average, as shown in Figure 11. Similarly, this transformation allows for performance enhancement as well, because it enables a class to take a more abstract view of the other class by accessing it via the added interface.

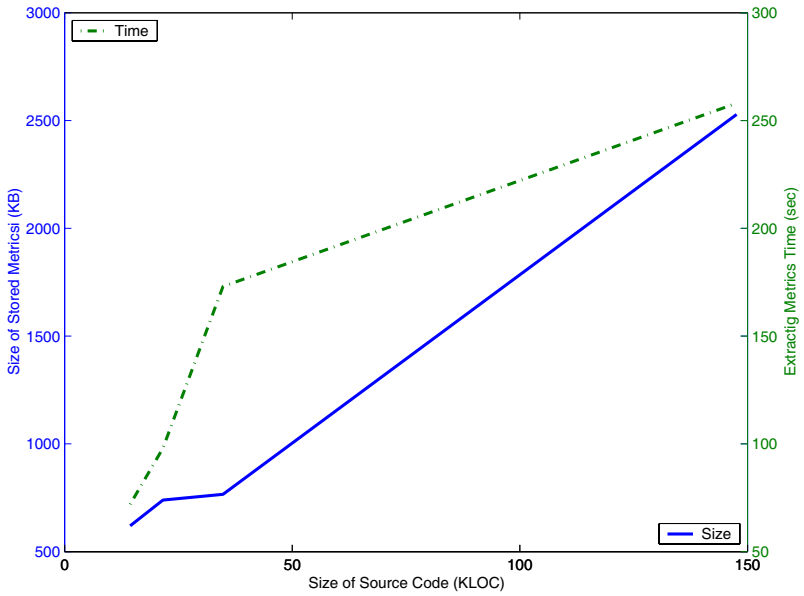


Figure 10. Graph of extracting metrics time and size results for the case studies.

Table VI. Impact of applied meta-pattern transformations on the OO metric suite for the case studies.

System	Transfor-mation	Metric name (%difference)										
		CDBC	CDE	DAC	DIT	LCOM	LD	NOC	NOM	RFC	TCC	WMC
JavaDoc	ABS	12.1	2.9	7.2	0	19.2	3.1	0.01	17.4	5.9	1.03	−9.07
JESS		9.7	2.1	3.6	0.02	11.4	24.7	0	3.6	2.6	5.7	−11.4
JavaDoc	EXT	9.1	17.5	−5.3	7.9	13.2	−9.8	3.4	5.8	0	6.7	0.01
JESS		12.4	7.6	−2.9	8.8	8.3	−5.5	12.5	4.8	0.01	18.3	0.03
JavaDoc	MOV	−11.4	0	0.01	2.6	11.3	0.02	1.1	6.9	−4.7	10.5	0.01
Ant		−8.3	0.04	0	8.2	17.6	0	1.1	4.1	−3.1	13.7	0.02
JavaDoc	ENC	5.4	23.7	1.8	15.4	−3.2	6.4	3.7	0	0.02	0	0.01
JESS		19.6	17.8	5.3	5.7	−10.6	1.1	13.9	0.07	0.03	0.01	0.02
JESS	BRL	25.6	3.6	0.02	0.04	−28.6	0	0.03	−7.8	7.8	0	15.8
JESS	WRP	33.2	42.9	0.02	0.05	−9.5	0.03	0	−14.1	0	17.4	−9.9
JDTCore		20.3	17.6	0	0.02	−17.4	0	0.02	−13.4	0.02	6.8	17.1

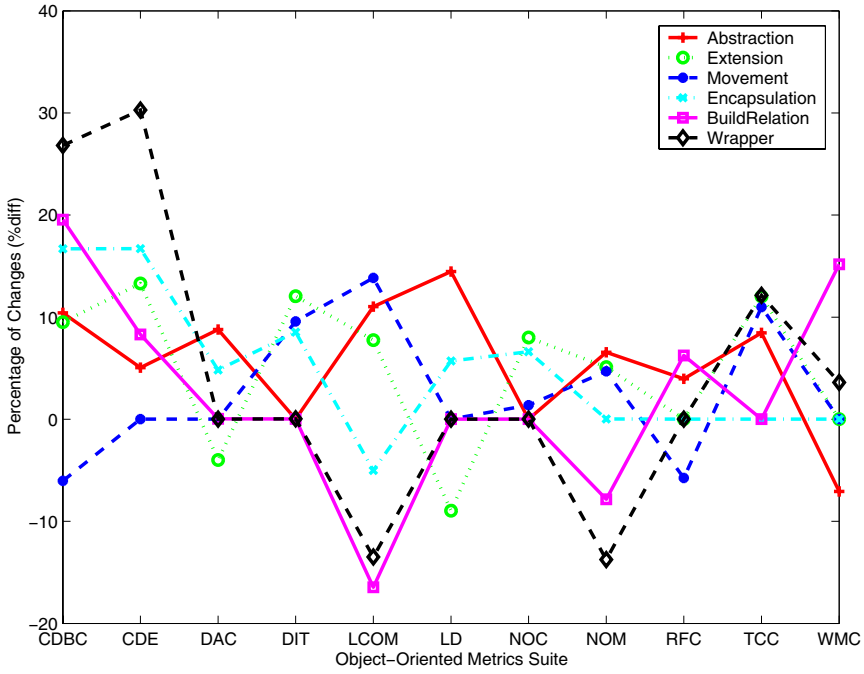


Figure 11. Impact of applying primitive transformations on maintainability.

High *CDE* [33] and *NOM* [36] metric values may indicate unnecessary complexity. Experimental results for the 'EXTENSION' transformation indicates an improvement for complexity by its positive impact on *CDE* [33] at the level of 15% on average, and on *NOM* [36] at the level of 7% on average, as shown in Figure 11.

Classes with many superclasses, yielding high *DIT* [29] and *NOC* [36] metric values, have their behavior scattered in many places, requiring more effort to maintain. This is the hidden cost of subclassing, often overlooked in OO designs. Experimental results for the 'ENCAPSULATION' transformation show an improvement for inheritance by its positive impact on these metric values at the level of 8% on average, as shown in Figure 11.

The cohesion of a class describes how closely the entities of a class (such as attributes and methods) are related. Often, cohesion is measured by establishing relationships between methods of the class in the case where the same instance variables are accessed. *TCC* [37] and *LCOM* [31] are proper indicators for this non-functional requirement. Experimental results for the 'MOVEMENT' meta-pattern transformation shows an improvement for cohesion by its positive impact on *TCC* at the level of 12% on average and on *LCOM* [31] at the level of 15% on average.

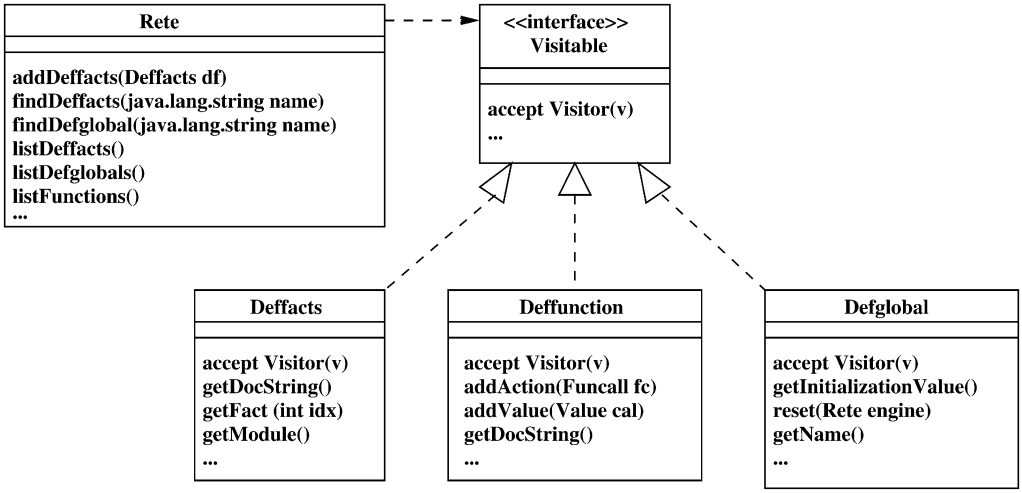


Figure 12. Part of the object model of the Java Expert System Shell (JESS).

11.3. Discussion on the proposed metric-based approach evaluation

In this section, one of the case studies is analyzed in more detail. Using the transformations described in Section 4, we have collected experimental results in order to evaluate the impact of particular implemented transformations. The results and the impact on maintainability by applying selected meta-pattern transformations for the case studies are illustrated in Figure 11.

We consider JESS. Three classes were detected by the assessment strategy as a deteriorated design from the maintainability point of view according to the OC2H rule. These three classes are called *Deffacts, Deffunction* and *Defglobal*. *Deffacts* is a public class which extends Java.lang.Object and implements java.io.Serializable. This part of the JESS system is depicted as a UML diagram in Figure 12. The *Rete* class is also the reasoning engine and executes the built Rete network, and coordinates many other activities. This is also a public class that extends java.lang.Object and implements java.io.Serializable.

One can create *deffact* objects and add them to a Rete engine using *Rete.addDeffacts()*. *Deffunction* is a public class which extends java.lang.Object and implements *Userfunction* and java.io.Serializable. One can create such objects and add them to a Rete engine using *Rete.addUserfunction*. *Defglobal* is a public class which extends java.lang.Object and implements java.io.Serializable. One can create *Defglobals* type objects and add them to a Rete engine using *Rete.addDefglobal*.

The values for the OO metrics suite of these classes are given in Table VII. To avoid the KCH rule applying for each of the three classes, we have to increase the value of *TCC*, to decrease the value of *DAC* and to decrease the value of *RFC*. As Figure 12 shows, the *Rete* class make use of the *Visitable* interface that has been implemented in three different implementation classes, namely *Deffacts*, *Deffunction* and *Defglobal*. The weakness of this structure becomes apparent if the programmer wants

Table VII. Object-oriented metrics for three classes of JESS.

Metrics	*Deffacts*	*Deffunction*	*Defglobal*
CDE	3.794	3.374	2.321
DAC	0	2	1
LCOM	9	5	5
NOM	9	9	5
RFC	30	15	24
TCC	11.3	25.7	34.6
WMC	16	30	50

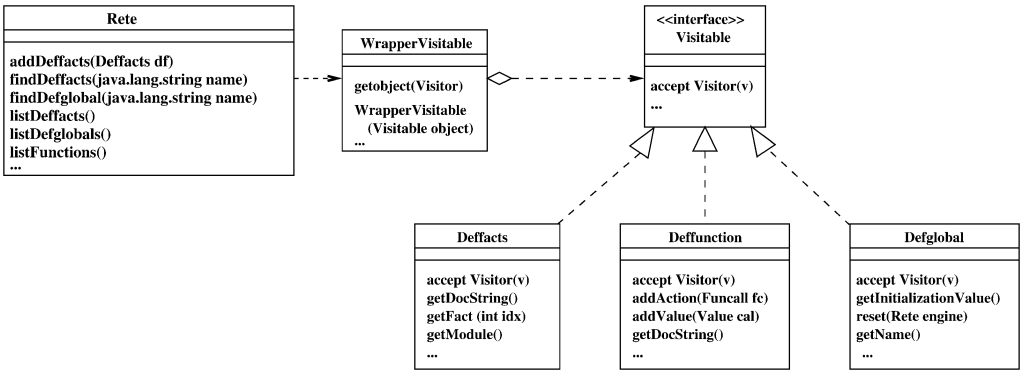


Figure 13. The same part of JESS after applying the 'WRAPPER' meta-pattern transformation.

to extend the interface in some way. Then, for each existing implementation class, a new class will have to be added.

After applying the SGE algorithm as discussed in Section 9, the three potential meta-pattern transformations are proposed to correct such design flaws.

The first candidate is a 'WRAPPER' meta-pattern transformation which enables the *Visitable* interface to be extended separately from its implementation. This transformation can create an abstract class for the three classes. As these three classes have common methods (such as accept, getDocString), the *RFC* and *DAC* do not change, which is sufficient to avoid the application of the WRAPPER meta-pattern transformation. This transformation is appropriate according to the context of the application.

In considering the WRAPPER meta-pattern transformation, it is clear that the theme of *delegation* is involved. A new wrapper class is to be added between the existing *Rete* class and the implementation classes. The effect of applying the WRAPPER transformation is depicted as a UML diagram in Figure 13. The duty of this class is to delegate all the requests it receives from the client (*Rete*) to the appropriate implementation object.

As illustrated in Figure 13, the WRAPPER meta-pattern transformation is used 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 object. This requires that all existing instantiations of the receiver class be also wrapped with an instantiation of the wrapper class itself. The overall effect of this meta-pattern transformation is to add a certain flexibility to the relationship between a client object and the receiver object it uses by increasing the value of *TCC* and decreasing the values of *LCOM* and *NOM*. All communication now goes via the wrapper object (*WrapperVisitable*), which means that run-time replacement of the receiver object classes becomes possible without the client object (*Rete*) being aware of the change. Experimental results confirm such maintainability improvement in terms of reducing the complexity and coupling measurement as shown in Table VI. A Similar situation was detected and applied for the Eclipse JDTCore as shown in Table VI. Both of these systems could be improved further by the application of Bridge design pattern transformation.

The second candidate can be the application of the EXTENSION meta-pattern transformation for these three classes in JESS, which proposes the creation of a set of specialized subclasses for each class. The three classes *Deffacts*, *Deffunction* and *Defglobal* are small and are already pretty much specialized, so this transformation and its corresponding path can be omitted.

Finally, the third candidate is the application of the ABSTRACTION meta-pattern transformation for these three classes of the JESS system. This cannot be performed because of a violation of the preconditions based on the source-code features for this part of system.

In this context, this work presents a metric-based approach to guide the choice of useful transformations. In the current experiments, the compiled catalogue of OO metrics and the proposed transformation framework allowed us to detect design flaws in a system and improve its design quality through applicable meta-pattern transformations.

12. RELATED WORK

Related work spans across several research areas, particularly OO reengineering and OO quality estimation. Basili *et al.* [39] and Briand *et al.* [40] showed that most of the metrics proposed by Chidamber and Kemerer [29] are useful to predict the fault-proneness of classes during the design phase of OO systems. In the same context, Li and Henry [36] showed that maintenance effort could be predicted from combinations of metrics collected from the source code of OO components.

The reengineering of OO software using transformations to improve its quality has been addressed by several researchers. Some techniques involving the decomposition of class hierarchy transformations in smaller modifications are proposed by Casais [51] and Opdyke [27]. In [51], a set of primitive update operations that can be used to decompose class modifications are enumerated. The completeness and correctness issues are presented, but not formally addressed. Similar work has been conducted by Opdyke [27]. He introduced the notion of behavior-preserving transformations named *refactorings*. A set of low-level refactorings is used to decompose high-level refactorings without introducing new errors into the system and modifying the program behavior. Preservation of the program behavior for each low-level refactoring is guaranteed when some preconditions are verified. A tool called *The Refactoring Browser* [52] was created using these transformations in the SmallTalk environment. Recently, Tokuda and Batory [53] indicated that programs can automatically

be reengineered using design patterns. In this work, the authors proposed transformations that can implement some design patterns. Most of the efforts in this research direction are concentrated on the definition of transformations and their implementation.

Several authors have addressed the particular problem of class hierarchy design and maintenance. In their work, transformations are typically used to abstract common behavior into new classes. Work in the context of the Demeter System has addressed the design of class hierarchies using an optimization process [7]. The objective function used in the optimization process is a global class hierarchy metric that measures the overall complexity of the class hierarchy. This work is therefore a first step in using metrics to guide the choice of useful transformations. Godin and Mili [54] proposed the use of concept (Galois) lattices and derived structures as formal frameworks for dealing with class hierarchy design or reengineering that guarantee maximal factorization of the common properties including polymorphism. The GURU tool [8] deals with the refactoring of methods and class hierarchy in an integrated manner.

More recently, some techniques have been developed with the specific goal of identifying restructuring opportunities. Kang and Bieman [55] sketched an approach to address the restructuring of the programs in a procedural paradigm. Their restructuring operations are applied to two design-level models in the form of graphs, namely the input–output dependence graph (IODG) and module interconnection graph (MIG), and decisions are guided by objective criteria. Simon *et al.* [56] presented a generic approach to generate visualizations supporting the developer to identify candidates for only four refactorings: move method, move attribute, extract class, and in-line class based on a metrical distance measure between two entities which supports the measurement of cohesion. Tourwé and Mens [57] tackled a similar problem by using logic meta programming (LMP) as a technique to support state-of-the-art software development. The main difference between our metric-based approach and their logic-based approach is that metrics are subject to interpretation whereas their detection technique is strict. The approaches are thus clearly complementary, since some bad smells are subject to interpretation as well, whereas others are more strict.

13. CONCLUSIONS

This paper presented a *reengineering process model* and a *transformation framework*. The process model focuses on the specification of soft-goal requirements for the target migrant system and a list of software transformations that have a positive impact on such requirements.

The reengineering framework focuses on the identification of error-prone code using metrics and the selection of the appropriate transformations that have the potential to enhance the target qualities and requirements for the new system. Specifically, we have investigated the use of OO metrics for detecting potential design flaws and for suggesting potentially useful transformations for correcting them.

Initial experiments with this reengineering strategy have demonstrated the feasibility of the approach and its usefulness. Also, the experiments indicate that the approach can help a designer or programmer to repair or maintain a software system in an incremental and localized way by suggesting proper software transformations. Furthermore, this strategy can be used to prevent loss of maintainability during evolution altogether or restore it through reengineering.

A direction for future work is to investigate the use of metrics with context and domain specific information. This may enable us to refine the selection of appropriate transformations by eliminating those that are not relevant or do not contribute towards the selected qualities being improved.

ACKNOWLEDGEMENTS

This work was funded by IBM Toronto Laboratory—Center for Advanced Studies (CAS), and by the Natural Sciences and Engineering Research Council (NSERC) of Canada. The authors wish to thank the anonymous reviewers whose constructive criticism and feedback helped to improve this manuscript.

REFERENCES

1. Tahvildari L, Gregory R, Kontogiannis K. An approach for measuring software evolution using source code features. *Proceedings IEEE Asia-Pacific Software Engineering (APSEC)*, Aoyama M, Poo DCC (eds.). IEEE Computer Society Press: Los Alamitos CA, 1999; 10–17.
2. Tahvildari L, Kontogiannis K, Mylopoulos J. Quality-driven software reengineering. *Journal of Systems and Software, Special Issue on: Software Architecture—Engineering Quality Attributes* 2003; **66**(3):225–239.
3. Tahvildari L, Kontogiannis K. On the role of design patterns in quality-driven reengineering. *Proceedings 6th European Conference on Software Maintenance and Reengineering (CSMR'02)*, Gyimothy T (ed.). IEEE Computer Society: Los Alamitos CA, 2002; 230–240.
4. Tahvildari L, Kontogiannis K. A software transformation framework for quality-driven object-oriented reengineering. *Proceedings International Conference on Software Maintenance (ICSM 2002)*, Antoniol G, Baxter I (eds.). IEEE Computer Society: Los Alamitos CA, 2002; 596–605.
5. Brown W, Malveau R, McCormick H III, Mowbray T. *Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley: Chichester, 1998.
6. Ciupke O. Automatic detection of design problems in object-oriented reengineering. *Proceedings IEEE Technology of Object-Oriented Languages and Systems (TOOLS)*, Firesmith D (ed.). IEEE Computer Society Press: Los Alamitos CA, 1999; 18–32.
7. Lieberherr KJ, Bergstein P, Silva-Lepe I. From objects to classes: Algorithms for optimal object-oriented design. *IEEE Journal of Software Engineering* 1991; **6**(4):205–228.
8. Moore I. Automatic inheritance hierarchy restructuring and method refactoring. *Proceedings of the ACM 11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Coplien J (ed.). ACM Press: New York, 1996; 235–250.
9. Riel AJ. *Object-Oriented Design Heuristics*. Addison-Wesley: Reading MA, 1998.
10. Woods SG, Quilici AE, Yang Q. *Constraint-Based Design Recovery for Software Re-engineering*. Kluwer: Boston MA, 1998.
11. Fowler M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley: Reading MA, 1999.
12. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley: Reading MA, 1995.
13. Baxter I, Pidgeon C. Software change through design maintenance. *Proceedings International Conference on Software Maintenance*, Harrold MJ, Visaggio G (eds.). IEEE Computer Society Press: Los Alamitos CA, 1997; 250–259.
14. Finnigan PJ, Holt RC, Kalas I, Kerr S, Kontogiannis K, Müller HA, Mylopoulos J, Perelgut SG, Stanley M, Wong K. The software bookshelf. *IBM Systems Journal* 1997; **36**(4):564–593.
15. Miller HW. *Re-engineering Legacy Software Systems*. Digital Press: Boston MA, 1998.
16. Patil P. Migration of procedural systems to object oriented architectures. *Master's Thesis*, University of Waterloo, Waterloo, Ontario, Canada, 1999.
17. Sneed H, Nyary E. Down-sizing large application programs. *Journal of Software Maintenance: Research and Practice* 1994; **6**(5):105–116.
18. Chung LK, Nixon BA, Yu E, Mylopoulos J. *Non-Functional Requirements in Software Engineering*. Kluwer: Norwell MA, 2000.
19. Aho AV, Sethi R, Ullman J. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley: Reading MA, 1988.
20. Garlan D, Kaiser GE, Notkin D. Using tool abstraction to compose system. *IEEE Computer* 1992; **25**(6):30–38.
21. Oman P, Hagemeister J. Constructing and testing of polynomials predicting software maintainability. *The Journal of Systems and Software* 1994; **24**(3):251–266.
22. Parnas DL. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 1972; **15**(12):1053–1058.
23. Tahvildari L. Quality-driven object-oriented reengineering framework. *Doctoral dissertation*, University of Waterloo, Waterloo, Ontario, Canada, 2003.

24. Pree W. Meta patterns—a means for capturing the essentials of reusable object-oriented design. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (*Lecture Notes in Computer Science*, vol. 821), Tokoro M, Pareschi R (eds.). Springer: Heidelberg, 1994; 150–162.
25. Braude EJ. *Software Engineering: An Object-Oriented Perspective*. Addison-Wesley: Reading MA, 2001.
26. Bauer M. Analyzing software systems using combinations of metric. *Proceedings of the ECOOP Workshop on Experiences in Object-Oriented Re-Engineering* (*Lecture Notes in Computer Science*, vol. 1743), Moreira A, Demeyer S (eds.). Springer: Heidelberg, 1999; 170–171.
27. Opdyke W. Refactoring object-oriented framework. *Doctoral dissertation*, University of Illinois, 1992.
28. Tahvildari L, Singh A. Categorization of object-oriented software metrics. *Proceedings IEEE Canadian Conference on Electrical and Computer Engineering*, El-Hawary ME (ed.). IEEE Service Center: Piscataway NJ, 2000; 235–239.
29. Chidamber SR, Kemerer CF. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering* 1994; **20**(6):476–493.
30. Etzkorn LH, Bansiya J, Davis C. Design and code complexity metrics for OO classes. *Journal of Object Oriented Programming* 1999; **12**(1):35–40.
31. Henderson-Sellers B. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall: Englewood Cliffs NJ, 1996.
32. McCabe T. A complexity measure. *IEEE Transactions on Software Engineering* 1976; **2**(4):308–320.
33. Bansiya J, Davis C, Etzkorn L. An entropy-based complexity measure for object-oriented designs. *Theory and Practice of Object Systems* 1999; **5**(2):111–118.
34. Hitz M, Montazeri B. Chidamber and Kemerer's metrics suits: A measurement theory perspective. *IEEE Transactions on Software Engineering* 1996; **22**(4):267–271.
35. Hitz M, Montazeri B. Measuring coupling in object-oriented systems. *Object Currents* 1996; **1**(4):124–136.
36. Li W, Henry S. Object-oriented metrics that predict maintainability. *Journal of Systems and Software* 1994; **23**(2):111–122.
37. Bieman JM, Kang BK. Cohesion and reuse in an object-oriented system. *Proceedings of the ACM SIGSOFT Symposium for Software Reusability, ACM SIGSOFT Software Engineering Notes*, Samadzadeh MH, Zand MK (eds.). ACM Press: New York, 1995; 259–262.
38. Etzkorn LH, Davis C, Li W. A practical look at the lack of cohesion in methods metric. *Journal of Object Oriented Programming* 1998; **11**(5):27–34.
39. Basili VR, Briand LC, Melo WL. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions of Software Engineering* 1996; **22**(10):751–761.
40. Briand LC, Morasca S, Basili VR. Defining and validating measures of object-based high-level design. *IEEE Transactions of Software Engineering* 1999; **25**(5):722–743.
41. Reengineering Tool for Java (RET4J). http://www.alphaworks.ibm.com/tech/ret4j [November 2001].
42. Datrix Metric Reference Manual, Version 4.1. http://www.iro.umontreal.ca/labs/gelo/datrix [August 2000].
43. Java Expert System Shell (JESS). http://herzberg.ca.sandia.gov/jess/ [November 2003].
44. Forgy CL. Rete: A fast algorithm for the many pattern/many objects match problem. *Artificial Intelligence* 1982; **19**(1):17–37.
45. NetBeans JavaDoc. http://javadoc.netbeans.org/ [February 2004].
46. NetBeans IDE. http://www.netbeans.org/ [February 2004].
47. Eclipse Ant Package. http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.ant.core/ [June 2002].
48. Apache Ant. http://ant.apache.org/ [February 2004].
49. Eclipse JDT Core Package. http://www.eclipse.org/documentation/html/plugins/api/org/eclipse/jdt/core/package-summary.html [June 2002].
50. Mamas E. Design and implementation of an integrated software maintenance environment. *Master's Thesis*, University of Waterloo, Waterloo, Ontario, Canada, 2000.
51. Casais E. An incremental class reorganization approach. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (*Lecture Notes in Computer Science*, vol. 162), Madsen L (ed.). Springer: Heidelberg, 1992; 114–132.
52. Roberts D, Brant J, Johnson R. A refactoring tools for SmallTalk. *Theory and Practice of Object Systems* 1997; **3**(4):253–263.
53. Tokuda L, Batory D. Evolving object-oriented designs with refactorings. *Proceedings IEEE 14th International Conference on Automated Software Engineering (ASE'99)*, Hall RJ (ed.). IEEE Computer Society Press: Los Alamitos CA, 1999; 174–181.
54. Godin R, Mili H. Building and maintaining analysis-level class hierarchies using galois lattice. *Proceedings of the ACM 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Paepcke A (ed.). ACM Press: New York, 1993; 394–410.
55. Kang BK, Bieman JM. A quantitative framework for software restructuring. *Journal of Software Maintenance: Research and Practice* 1999; **11**(4):245–284.

56. Simon F, Steinbrückner F, Lewerentz C. Metric based refactoring. *Proceedings 5th European Conference on Software Maintenance and Reengineering (CSMR'01)*, Sousa P, Ebert J (eds.). IEEE Computer Society Press: Los Alamitos CA, 2001; 30–38.
57. Tourwé T, Mens T. Identifying refactoring opportunities using logic meta programming. *Proceedings 7th European Conference on Software Maintenance and Reengineering (CSMR'03)*, van den Brand M, Gyimothy T (eds.). IEEE Computer Society Press: Los Alamitos CA, 2003; 91–100.

AUTHORS' BIOGRAPHIES

Ladan Tahvildari is an Assistant Professor in the Department of Electrical and Computer Engineering at the University of Waterloo, Canada. She received her BSc degree (with Honors) in the area of Software Engineering from Iran University of Science and Technology at the Department of Computer Engineering in 1991. Having worked in industry for six years, she returned to school and received her MASc and PhD from the University of Waterloo in 1999 and 2003, respectively. Her research interests include software engineering, software evolution, reverse engineering, quality-based software reengineering, and software architecture. She is the Program Co-Chair for STEP 2004 and the Workshop Chair for WCRE 2004. She was the Publicity Chair for WCRE 2003.

Kostas Kontogiannis is an Associate Professor in the Department of Electrical and Computer Engineering at the University of Waterloo, Canada. He is also a Visiting Scientist with the Centre for Advanced Studies at the IBM Toronto Laboratory. He received his PhD degree from McGill University in 1996. Together with his research group, he investigates technologies to migrate legacy software to object-oriented and network-centric platforms. His specific topics of interest include techniques and tools for source code representation, quality preserving source code transformations, and techniques for the integration of Web services. He was the recipient of the 2002 IBM University Partnership Program Award. He was the General Chair for STEP 2003, and the Program Co-Chair for IWPC 2001.