A Metric-Based Approach to Enhance Design Quality Through Meta-Pattern Transformations [*]

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

During the evolution of object-oriented legacy systems, improving the design quality is most often a highly demanded objective. For such systems which have a large number of classes and are subject to frequent modifications, detection and correction of design defects is a complex task. The use of automatic detection and correction tools can be helpful for this task. Various research approaches have proposed transformations that improve the quality of an object-oriented systems while preserving its behavior. This paper proposes a framework where a catalogue of object-oriented metrics can be used as indicators for automatically detecting situations where a particular transformation can be applied to improve the quality of an object-oriented legacy system. The correction process is based on analyzing the impact of various meta-pattern transformations on these object-oriented metrics.

1 Introduction

Design defects can be recognized in the early stages of software development or during system evolution. They cause the system to exhibit low maintainability, low reuse, high complexity and faulty behavior. Specifically, for object-oriented legacy systems which have been faced with frequent modifications, detection and correction of such design flaws is a complex task.

Our previous work on improving the quality of object-oriented legacy systems includes : i) using metrics for quality estimation [29, 33], and ii) proposing a software transformation framework based soft-goal dependency graphs to enhance quality [30, 31, 32]. Both aspects have been treated mostly 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 work, we identify a catalogue of object-oriented software metrics that can be related to object-oriented design properties. Then, we investigate the use of metrics for detecting potential design flaws and for suggesting potentially useful transformations for correcting them.

This paper is organized as follows. Section 2 proposes a classification of object-oriented design flaws which is a step towards discovering recurring detection and correction methods. Section 3 presents the proposed re-engineering strategy using object-oriented metrics to detect and correct design flaws while Section 4 discusses quality design heuristics to guide re-engineering strategy. Section 5 classifies a selection of object-oriented metrics which are of the interest for this research. Section 6 presents a diagnosis algorithm. Section 7 shows the impact of applying meta-patterns on metrics Finally, Section 8 discusses an application scenario of the proposed classification and Section 10 provides the conclusion and insights of future work.

2 A Classification of OO Design Flaws

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

[*]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.

A design could deteriorate for several reasons. A class may have conformed to good quality object-oriented design (*i.e.*, encapsulation, information hiding, data abstraction) in the initial stages of development and lost its integrity due to: i) addition of methods/data members, ii) a base class trying to accomplish too much for its derived classes, iii) a class attempting to handle too many different situations, grouping what should be several different derived classes into a single class, iv) a class with increasing number of relationships and associations with other classes and abstract data types.

By introducing a classification of design flaws, we aim to discover recurring detection and correction methods. This classification will ease the assessment of new re-engineering techniques and tools. Sorting and classifying design flaws is complex because of the multiple points of view available. We propose the following classification from the literature [7, 10, 15, 24, 36] based on the scope of design flaws inside an object-oriented applications. It 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 retain these three categories because they represent three distinct levels of abstraction and thus must rely on different detection and correction techniques.

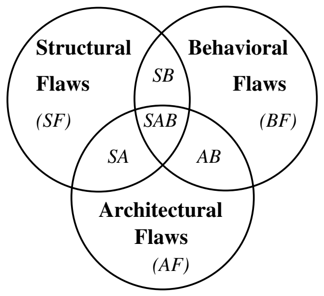

Figure 1. Design Flaws Classification.

These three categories are not orthogonal and several design flaws do not fit simply in a single category. We can define four additional categories as depicted in Figure 1 which are included the intersections of the three previous ones. Thus, evolving from the three main categories, we can introduce seven categories to classify the design flaws. These categories which allow a finer-grain classification of the design flaws are as follows:

- **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 [15].

- **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 is the algorithms overweight the data structure, then the data structure extension is slowed down because it must be modified every time a new algorithm is added and it is likely to grow rapidly out of control [16].

- **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 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 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. For example, the used of the *Composite Pattern* [16] when a single field could be used [33].

- **Intersection of SF and BF (SB)**
  This category encloses design flaws involving both the semantic of the class and its internal structure. There are some defects in the behavior of the class which corrections imply changing it structure as well as some defects in the internal structure of the class which corrections implies changing in its behavior.

- **Intersection of AF and BF (AB)**
  This category encompasses design flaws related to both architectural and behavioral of the classes. There are a set of design flaws related to the application architecture which corrections imply changing the semantics of its classes. For example, a "God" class [27] is a sign of a bad architecture which improvement implies changing the semantic of at least the "God" class. Also, there are some design flaws in the behavior of classes which corrections imply changes in their architecture.

- **Intersection of SF, AF, and BF (SAB)**
  The last category includes the set of all the design flaws implying the structure, semantics, and the architecture of the application.

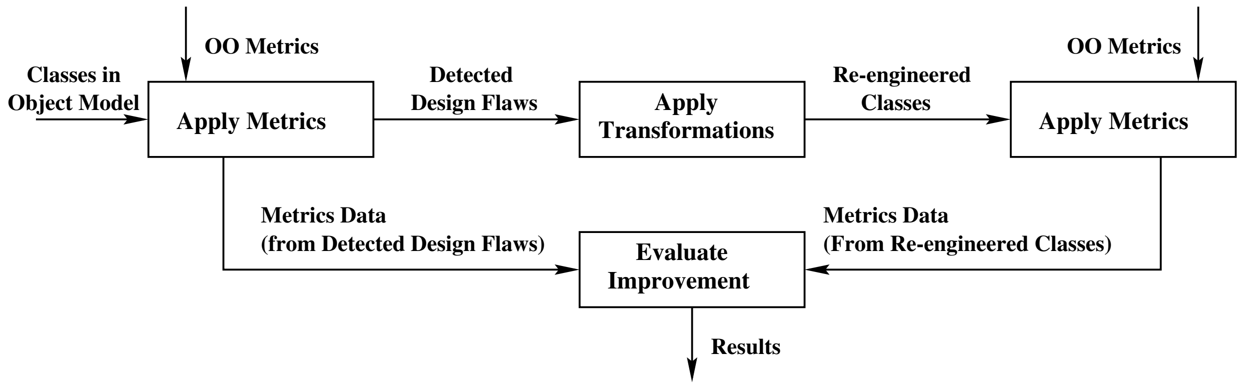

Figure 2. Re-engineering Strategy for Design Flaws.

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 differentiating among design flaws that root in one category and imply changes in another one. For example, duplicate code across classes is detected into internal structures (SF) of the classes, but resulting flaws appear in both internal structures (SF) and their architecture (AF).

Our concern for improving the quality of object-oriented design of legacy systems is related to apply the changes which preserve the behavior of the system. It means that the behavioral flaws are out of the scope of this research work. Due to this assumption, we only focus on SF, AF, and the intersections between them which can cause decreasing the design quality.

3 Design Flaws Re-engineering Framework

During the development of object-oriented legacy systems, an incremental approach is used [5, 26]. The initial design model created in the first increment may have good design properties. However, over subsequent increments, the quality may be deteriorated. It means that there is a risk that the class design will deteriorate in quality with each increment. Over time, such classes may become difficult to maintain and become prone to errors. Consequently, we need to devise a technique not only for detecting these design flaws but also for correcting them.

Figure 2 shows the proposed re-engineering strategy using object-oriented metrics. As known, an object model has several levels of representation, including *application level*, *subsystem level*, *class level*, and *function level*. While design flaws can occur at any level, our focus here is on class deterioration. We believe that this is the most fundamental level that constitutes a system. Improving deteriorated classes should help to keep object-oriented legacy systems operational. After extracting an object model at class level through reverse engineering, the proposed strategy as depicted in Figure 2 makes use of the the following steps :

- **Step 1 :** To measure and record the object-oriented metrics in order to detect classes for which quality has deteriorated. While there are several reasons for a design to 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 object-oriented design metrics which relate to different categories of quality design heuristics. In Section 5, we will propose and discuss this useful catalogue.

- **Step 2 :** To re-engineer detected design flaws using proper transformations. For correcting such design flaws through software transformations, we need to study the impact of a predefine set of transformations on a predefined set of metrics. In Section 7, we not only select the transformation that are of our interest in this research work but also discuss their impact on the predefined catalogue of object-oriented metrics.

- **Step 3 :** To re-apply and record the same object-oriented metrics to the re-engineered classes and finally compare the recorded results to evaluate design improvement. Once the transformation and the role of the class are determined, it is necessary to verify that the transformation makes sense in the particular context of the application. Based on the preconditions for each transformation, we can find all possible transformations based on the source code features that can be applied when our proposed framework detects a symptomatic situation.

4 Quality Design Heuristics

One way to detect design flaws at the class level is to identify violations of a “good” object-oriented software design by performing source code analysis. However, some guidelines and principles exist [27] to build a “good” design, but there is no consensus on what a “good” design really is.

While there are several reasons for a class to lose quality over time, here the focus is on the classes that have high coupling and low cohesion. These characteristics often result in loss of abstraction and encapsulation. They are those highly coupled classes that often loose cohesion during the course of development. Based on this assumption, the quality design heuristics will be proposed to detect design flaws at the class level. The following subsections will discuss in detail these concepts. In a similar manner, other design principles can be checked and violations can be detected by using different heuristics [27]. Due to the lack of space, we focus in this paper only on the two of these heuristics.

4.1 Design Heuristic 1 : Key Classes

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* [3] which can be characterized by the specific properties. These classes which we called them as *key classes* manage a large amount of 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. Finding these classes is a starting point in our framework to detect potential design flaws and correct them properly based on meta-pattern transformations. Figure 3 illustrates such an analysis. The classes of the object-oriented 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 the two metrics by computing the *distance* d of a class from the origin of the coordinate system. If x denotes the complexity value of a class, and y its coupling value, we compute the combined value d as :

$$d = \sqrt{x^2 + y^2}$$

In some cases, if the metrics use a very different scale, some normalization might be required. We can then use the following formula :

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

This combined value allow us to compare classes. Classes with higher values for d are better candidates to be considered 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 needs to be taken care of.

4.2 Design Heuristic 2 : One Class-One Concept

A very basic principle in object-oriented software engineering states that a class should implement one single concept of the application domain. Some violations of this principle can be detected by using the following assumptions :

- A class that implements more than one concept, has probably low cohesion measurements, since these concepts can be implemented separately.
- 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 object-oriented 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 [25]. 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.

5 Object-Oriented Metrics Suite

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

In this section, we present object-oriented metrics defined in the literature and can be used to assess object-oriented system qualities. These metrics fall into different

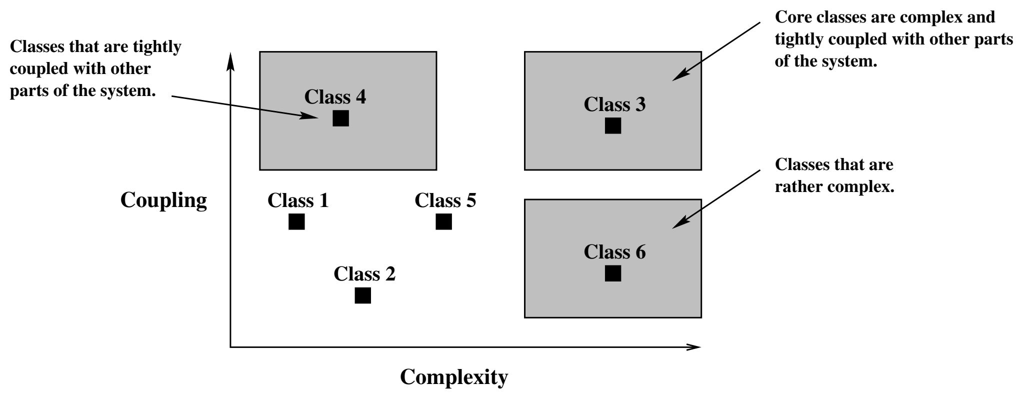
Classes that are tightly coupled with other parts of the system.
Core classes are complex and tightly coupled with other parts of the system.
Class 4
Class 3
Classes that are rather complex.
Coupling
Class 1
Class 5
Class 6
Class 2
Complexity

Figure 3. Key Classes in an Object-Oriented Legacy System.

categories depending on the aspects of design flaws that we plan to detect and correct them in the proposed design flaws re-engineering framework. The proposed selection of the object-oriented metrics are classified in three major categories [34] : complexity metrics, coupling metrics, and cohesion metrics as depicted in Table 1 on the alphabetic order of the metric names and described as follows.

- **Complexity Metrics**
  We consider those metrics because they may give us 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) measures the complexity of a class by adding up the complexities of the methods defined in the class [9, 12]. A special case of *WMC* (which is very simple to compute) is *NOM* (Number Of Methods) which measures the complexity of a class by counting the number of methods defined in that class [18] as well which is called *RFC* (Response Set For A Class) metric [9]. 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 LOC or the McCabe Cyclomatic complexity [23]. Class Definition Entropy (CDE) [1] metric identifies complex classes in an object-oriented 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 re-engineering 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 object-oriented 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 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 a good starting point when trying to understand an unfamiliar object-oriented legacy system. Analyzing the viewpoints suggested for the different coupling metrics, we are able to remark the reuse degree and maintenance effort for a class are decisively influence by the coupling level between classes. *DAC* (Data Abstract Coupling) measures coupling between classes that results from attribute declarations [19, 20, 21]. *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).

- **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) [4, 13, 19, 20] 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

Metric	Name	Category	Definition
CDE [1]	Class Definition Entropy	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 declaration. Consider n_1 is the number of unique name strings used in the class definition, N_1 is the total number (non-unique) of name strings used in the class definition, f_i for $1 \leq i \leq n_1$ is the frequency of occurrence of the i^{th} name string used in the class, then $CDE = -\sum_{i=1}^{n_1}(f_i/N_1)log(f_i/N_1)$.
DAC [21]	Data Abstraction Coupling	Coupling	DAC is the number of ADT's defined in a class.
LCOM [18]	Lack of Cohesion in Methods	Cohesion	Consider a class C, its set M of m methods $M_1, ..., M_m$, and its set A of a data members $A_1, ..., A_a$ 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, $LCOM(C(M,A)) = \frac{\left(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)\right)-m}{1-m}$.
NOM [21]	Number Of Methods	Complexity	NOM is the number of local methods defined in a class which may indicate the operation property of a class
RFC [9]	Response For a Class	Complexity 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. Mathematically it can be defined using elements of set theory, as : $RS = \{M\} \cup_i \{R_i\}$ where $\{R_i\}$ is the set of methods called by method i and $\{M\}$ is the set of all methods in the class. RFC is the cardinality of the response set for that class. Then, mathematically $RFC = \lvert RS \rvert$.
TCC [4]	Tight Class Cohesion	Cohesion	A class can be represented as a collection of abstract methods (AM) where each AM corresponds to a visible method in the class. The representation of a class using abstract method is called an abstracted class (AC) which is a multi-set and can be formally expressed as : $AC(C) = \lVert AM(M) \vert M\ V(C) \rVert$ where $V(C)$ is the set of all visible methods in a class C and in the ancestors of C. Let $NP(C)$ to be the total number of pairs of abstract methods in $AC(C)$. Let $NDC(C)$ to be the number of directed connection in $AC(C)$, then TCC is the relative number of directly connected methods which can be expressed as : $TCC(C) = NDC(C)/NP(C)$.
WMC [9]	Weighted Methods per Class	Complexity	Consider a class C_1, with methods $M_1, ..., M_n$ and $c_1, ..., c_n$ are the static complexity of the methods, then : $WMC = \sum_{i=1}^{n} c_i$.

Table 1. Selected Object-Oriented Metrics.

common instance variable. In the literature, several formulas have been introduced to compute Lack of Cohesion [9, 18, 21]. We adopted a definition in [18] 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 variable.

6 A Diagnosis Algorithm

Based on the discussion in Section 3, a diagnosis algorithm shown in Figure 4 summarizes the detection and correction activities as implemented by our proposed design flaws re-engineering framework.

The first step is to apply the *key classes (DH1)* rule by using both a 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. Therefore, monitoring *DAC* values can be promising. When a significant number of classes evolves to higher *DAC* measurements, some *refactoring operations* [15] or meta-pattern transformations [32, 30] of the system could be appropriate, to reduce coupling.

Good object-oriented design styles usually require that classes have high cohesion, since they should encapsulate concepts that belong together. Classes with low co-

```
AC is a set of classes in an object model
   extracted from a legacy system

  FOR each class C of AC do

    - Calculate the metric values from
      the predefined catalogue

    - Apply quality design heuristics
      to detect design flaws

    - IF it is a deteriorated class

        + Select potential transformations
          that can correct the flaws

        + Apply the transformations that
          corresponnd to the context of
          C because of its features

    - End IF

  EndFOR
```

Figure 4. Description of Diagnosis Algorithm.

hesion often represent violations to a flexible, extensible or reusable design. All of these are issues that must be dealt with during the object-oriented re-engineering process. Therefore, by applying cohesion metrics like *TCC* and coupling metrics like *DAC* and *RFC* to the object-oriented legacy system, possible violations of the principle *one class - one concept (DH2)* rule can be found. These classes tend to have either low *TCC* values or high *DAC* and high *RFC* values. For example, classes that have very low *TCC* values, can often be split [15]. 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 object-oriented way. These classes are not implementing a self contained object from the application domain, they just group methods together, acting as a module. In a similar manner, other design principles can be checked and violations can be detected by using metrics [27].

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 was split into more classes with individual higher cohesion. The LCOM metrics helps to identify such design flaws.

7 Impact of Applying Meta-Patterns on Metrics

Once a symptomatic situation is detected using object-oriented metrics, the next step is to propose possible transformations that improve the quality of the program while preserving its behavior. In Sections 4, 5, and 6, we established a cause-to-effect relationship between some combinations of metrics and a poor design quality. It means that we showed that by changing the code to improve the values of certain metrics, we presume that we can also improve the quality of an application or program. The problem to solve now is how the code should be changed to improve the corresponding metrics. An intuitive solution is to find out which transformation (or a set of transformations) allows changing the value of a particular metric (or a set of metrics). To respond to such a kind of 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 the external structures of the classes, and ii) analyze the impact of each transformation on the predefined set of metrics.

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 our context, meta-pattern transformations are changes in the design whose purpose is to improve the quality of a system while preserving its behavior. For this work, we use the meta-pattern transformations proposed in [32, 30]. These transformations modify the structure of a program which will possibly modify the values of the metrics. 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 meta-pattern transformation on metrics for the classes involved is shown in Table 2. Note that '$+$' means that there is a positive impact, '$-$' means that there is a negative impact, and 'NI' means that there is no impact.

8 A Case Study: Java Expert System Shell

To illustrate the approach proposed in this paper, we present an example of the application of the diagnosis algorithm in this section. We apply the proposed re-engineering strategy for design flaws on Java Expert System Shell (JESS) [1].

JESS is a rule engine and scripting environment written entirely in Sun's Java language by Ernest Friedman-Hill at Sandia National Laboratories [2]. 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 "reason" using knowledge you supply in the form of declarative rules. JESS is surprisingly fast, and for some problems is faster than CLIPS itself. 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 [14] to process rules,

[1] http://herzberg.ca.sandia.gov/jess/
[2] http://www.sandia.gov/

Meta-Pattern Name	Description	CDE	DAC	LCOM	NOM	RFC	TCC	WMC
ABSTRACTION	adds an interface to a class which enables another class to take a more abstract view of the first class by accessing via interface	$-$	NI	NI	$+$	$+$	$-$	NI
EXTENSION	constructs an abstract class from an existing class and creates an extends relationship between the two classes	NI	$+$	$+$	NI	$+$	$+$	$-$
MOVEMENT	moves parts of an existing class to a component class and sets up a delegation relationship from the existing class to its component	$-$	$+$	$+$	NI	$-$	$+$	$-$
ENCAPSULATION	weakens the association between two classes by packaging the object creation statements into dedicated methods	$-$	$+$	$+$	NI	$-$	$+$	$+$
BUILDRELATION	operates the relationship between the classes in a more abstract fashion via an interface	$+$	$+$	NI	NI	$-$	$+$	$+$
WRAPPER	wraps 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	$+$	$+$	NI	NI	$+$	$+$	$-$

Table 2. Impact of the Meta-Pattern Transformations on OO Metrics Suite.

a very efficient mechanism for solving the difficult many-to-many matching problem. JESS adds many features to CLIPS, including backwards chaining and the ability to manipulate and directly reason about Java objects. Jess is also a powerful Java scripting environment, from which one can create Java objects and call Java methods without compiling any Java code.

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

Table 3. Object-Oriented Metrics for Three classes of JESS.

We use the Re-engineering Tool Kit for Java [3] for extracting facts from the source code in order to provide a high-level view of the systems. For our analysis approach 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 [11].

Three classes were detected by the assessment strategy as a bad design from the maintainability point of view according to one class-one concept heuristic. These three classes are called respectively, *Deffacts*, *Deffunction*, and *Defglobal*. *Deffacts* is a public class which 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 object-oriented metrics suite of these classes are given in Table 3. To avoid that the DH2 rule applies 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. From Table 2, we can suggest to apply "WRAPPER" meta-pattern transformation. This can create an abstract class for the three classes. As these three classes have three common methods (accept, getname, tostring), the NOM and LCOM do not change which is sufficient to avoid the application of "WRAPPER" meta-pattern trans-

[3]http://www.alphaworks.ibm.com/tech/ret4j

formation. This transformation is appropriate according to the context of the application.

Another suggestion can be the application of "EXTENSION" meta-pattern transformation which proposes to create a set of specialized subclasses for each class. The three classes *Deffacts*, *Deffunction*, and *Defglobal* are small and are already pretty much specialized, then this correction for removing the design defects can be rejected.

9 Related Work

Related work cuts across several research areas and particularly object-oriented re-engineering and object-oriented quality estimation. Basili *et al.* [2] and Briand *et al.* [6] showed that most of the metrics proposed by Chidamber and Kemerer [9] are useful to predict fault-proneness of classes during the design phase of object-oriented systems. In the same context, Li and Henry [21] showed that maintenance effort could be predicted from combinations of metrics collected from source code of OO components.

Re-engineering of object-oriented software using transformations to improve its quality has been address by several researchers. Some techniques involving decomposition of class hierarchy transformations in smaller modifications are proposed by Casais [8] and Opdyke [25]. In [8], 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 [25]. 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 in 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* [28] was created using these transformations in the SmallTalk environment. Recently, Tokuda and Batory [35] show that programs can automatically re-engineered using design patterns. In this work, authors proposed transformations that can implement some design patterns. Most of the efforts in this research directions concentrated on the definition of transformations and their implementation. To the best of our knowledge, there is no effort on the automatic detection of the situation where these transformations can apply.

Several authors have addressed the particular problem of class hierarchy design and maintenance. In their works, transformations are used typically 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 [22]. 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 [17] proposed the use of concept (Galois) lattices and derived structures as of formal framework for dealing with class hierarchy design or re-engineering that guarantees maximal factorization of the common properties including polymorphism. The GURU tool [24] dealt with refactoring of methods and class hierarchy in an integrated manner.

However, not much effort has been invested for systematically documenting object-oriented metrics as a guide not only for detection design flaws in the reverse engineering process but also for correction it through proper transformations in the forward engineering process. In this context, the catalogue of object-oriented metrics and the proposed transformation framework allow for achieving specific quality requirements in the migrant system.

10 Conclusion

This paper addressed a process activity that can detect and rectify design defects from a practical point of view. This comprehensive process model focused on class deterioration primarily due to lack of cohesion induced by high coupling which is prevalent in object-oriented legacy systems.

We have investigated the use of metrics for detecting potential design flaws and for suggesting potentially useful transformations for correcting them. Initial experiments with this re-engineering strategy have demonstrated the feasibility of the approach and its usefulness. Indeed, our approach can help a designer or programmer by suggesting proper meta-pattern transformations. This also help to focus on a particular part of a large system.

We believe that the explicit nature of our proposed framework is very useful to develop and maintain good quality object-oriented systems. This strategy can be used to prevent loss of maintainability or restore it through reengineering.

A direction that we will explore in our future work is to associate the metrics to be used with context and appropriate domains. This will enable us to refine the suggestions by eliminating those that are not relevant.

References

[1] J. Bansiya, C. Davis, and L. Etzkorn. An entropy-based complexity measure for object-oriented designs. *Theory and Practice of Object Systems*, 5(2):111–118, 1999.

[2] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions of Software Engineering*, 22(10):751–761, October 1996.

[3] M. Bauer. Analyzing software systems using combinations of metrics. In O. Ciupke and S. Ducasse, editors, *Proceedings of the ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*, May 1999.

[4] J. Bieman and B. Kang. Cohesion and reuse in an object-oriented system. In *Proceedings of ACM Symposuim for Software Reusability (SSR)*, pages 259–262, 1995.

[5] E. J. Braude. *Software Engineering: An Object-Oriented Perspective*. Addison-Wesley, 2001.

[6] L. C. Briand, S. Morasca, and V. R. Basili. Defining and validating measures of object-based high-level design. *IEEE Transactions of Software Engineering*, 25(5):722–743, September/October 1999.

[7] K. Brown. Design reverse-engineering and automated design pattern detection in smalltalk. Technical report tr-96-07, University of Illinois at Urbana-Champaign, 1996.

[8] E. Casais. An incremental class reorganization approach. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS 615, Springer*, pages 114–132, Utrecht, The Netherlands, June 1992.

[9] S. R. Chidamber and C. F. Kemerer. A metric suite for object-oriented design. *IEEE Transactions of Software Engineering*, 25(5):476–493, June 1994.

[10] O. Ciupke. Automatic detection of design problems in object-oriented re-engineerin. In *Proceedings of the IEEE Technology of Object-Oriented Languages and Systems - TOOLS 30*, pages 18–32, 1999.

[11] Datrix metric reference manual, version 4.1. Bell Canada, 2000. Also available at http://www.iro.umontreal.ca/labs/gelo/datrix.

[12] L. H. Etzkorn, J. Bansiya, and C. Davis. Design and code complexity metrics for oo classes. *Journal of Object Oriented Programming*, 12(1):35–40, March 1999.

[13] L. H. Etzkorn, C. Davis, and W. Li. A practical look at the lack of cohesion in methods metric. *Journal of Object Oriented Programming*, 11(5):27–34, September 1998.

[14] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

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

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

[17] R. Godin and M. Hafedh. Building and maintaining analysis-level class hierarchies using galois lattice. In *Proceedings of the ACM 8^{th} Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 394–410, Washington, DC, September 1993.

[18] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1996.

[19] M. Hitz and B. Montazeri. Chidamber and kemerer's metrics suits: A measurement theory perspective. *IEEE Transactions on Software Engineering*, 22(4):267–271, April 1996.

[20] M. Hitz and B. Montazeri. Measuring coupling in object-oriented systems. *Object Currents*, 1(4), 1996.

[21] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993.

[22] K. J. Lieberherr, P. Bergstein, and I. Silva-Lepe. From objects to classes: Algorithms for optimal object-oriented design. *IEEE Journal of Software Engineering*, 6(4):205–228, 1991.

[23] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[24] I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings of the ACM 11^{th} Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 235–250, San Jose, California, October 1996.

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

[26] V. T. Rajlich and K. H. Bennett. A staged model for the software life cycle. *IEEE Computer*, 33(7):66–71, 2000.

[27] A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[28] D. Roberts, J. Brant, and R. Johnson. A refactoring tools for smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[29] 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.

[30] L. Tahvildari and K. Kontogiannis. A methodology for developing transformations using the maintainability soft-goal graph. In *Proceedings of the IEEE 9^{th} International Working Conference on Reverse Engineering (WCRE)*, pages 77–86, Richmond, Virginia, November 2002.

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

[32] L. Tahvildari and K. Kontogiannis. A software transformation framework for quality-driven object-oriented re-engineering. In *Proceedings of the IEEE International Conference on Software Maintenence (ICSM)*, pages 596–605, Montreal, Canada, October 2002.

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

[34] L. Tahvildari and A. Singh. Categorization of object-oriented software metrics. In *Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering*, pages 235–239, Halifax, Nova Scotia, May 2000.

[35] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. In *Proceedings of the IEEE 14^{th} International Conference on Automated Software Engineering (ASE)*, pages 174–181, Cocoa Beach, Florida, October 1999.

[36] S. G. Woods, A. E. Quilici, and Q. Yang. *Constraint-Based Design Recoveryfor Software Re-engineering*. Kluwer Academic Publishers, 1998.