Can Metrics Help Bridging the Gap Between the Improvement of OO Design Quality and Its Automation?*

Houari A. Sahraoui
DIRO, Université de Montréal
C.P. 6128, succ. CV,
Montreal (QC)
Canada H3C 3J7
sahraouh@iro.umontreal.ca

Robert Godin
Université du Québec à Montréal
C.P.8888, Succ.CV,
Montreal (QC), Canada
H3C 3P8
godin.robert@uqam.ca

Thierry Miceli
Pixel Systems Inc.
4750 Henri-Julien,
Montréal (Québec) Canada
H2T 2C8
tmiceli@sympatico.ca

Abstract

During the evolution of object-oriented systems, the preservation of correct design should be a permanent quest. However, for systems involving a large number of classes and subject to frequent modifications, detection and correction of design flaws may be a complex and resource-consuming task. The use of automatic detection and correction tools can be helpful for this task. Various work propose transformations that improve the quality of an object-oriented system while preserving its behavior. In this paper we propose to investigate whether some object-oriented metrics can be used as indicators for automatically detecting situations where a particular transformation can be applied to improve the quality of a system. The detection process is based on analyzing the impact of various transformations on these object-oriented metrics using quality estimation models.

1. Introduction

Design flaws, introduced in early stages of the development or during system evolution, are a frequent cause of low maintainability, low reuse, high complexity and faulty behavior of the programs [19]. The preservation of correct design should be a permanent quest. However, for large systems subject to frequent modifications, detection and correction of design flaws may be a complex and resource consuming task Automated tools for assisting this process can help alleviate this task.

Previous work on improving the quality of object systems includes using metrics for quality estimation and automated transformations to improve quality. However both aspects have been treated mostly independently of 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 metrics.

As a first step, we analyzed formally the impact of several common transformations on several metrics. This knowledge is incorporated in our OO1 prototype corrector tool. The tool is used to help improving the quality of C^{++} programs. The function of the tool is analogous to a linguistic assistant for a text processor. The tool computes several quality metrics on the source code. The metrics are used to detect potential design flaws. Based on these estimations, the tool suggests particular transformations that can be automatically applied in order to improve the quality as estimated by the metrics. Evidently, this should be seen as a heuristic process and, as for linguistic aids, the process may include some form of human intervention and acknowledgement before applying the suggested transformations. Although, our initial investigation has addressed OO program code, the same idea could be applied to earlier software design artifacts or to non-OO software.

The remainder of this paper is organized as follows. Section 2 surveys the related work in the area of software metrics and transformations. Section 3 gives an overview of the proposed technique. Section 4 describes the prototype tool and a case study. Section 5 presents our conclusion for this work.

2. Related work

Related work cuts across several research areas and particularly object-oriented software reengineering and OO quality estimation. For the case of OO software, Basili & al. show in [1] that most of the metrics proposed by Chidamber and Kemerer in [3] are useful to predict fault-

* This work was partly funded by CRIM Montreal.

proneness of classes during the design phase of OO systems. In the same context, Li and Henry showed that maintenance effort could be predicted from combinations of metrics collected from source code of OO components [9]. In [4], Demeyer and Ducasse show for the particular domain of OO frameworks, that size and inheritance metrics are not reliable to detect problems, but are good indicators for the stability of a framework. More recently our team proposed a set of models for different quality characteristics in [8], [11], [12] and [13]. The particularity of our work is that we use machine-learning techniques to build the estimation models. These techniques generate interesting results even with small-size learning sets.

Reengineering of OO software using transformations to improve its quality has been addressed by several researchers. Some techniques involving decomposition of class hierarchy transformations in smaller modifications are proposed by Casais and more recently by Opdyke. In [2], Casais enumerates a set of primitive update operations that can be used to decompose class modifications. The completeness and correctness issues are presented but not formally addressed. Similar work has been conducted by Opdyke (see [15] and [16]). He introduces 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 or 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* [18] was created using these transformations in the Smalltalk environment. Recently, Tokuda and Batory show that programs can be automatically reengineered using design patterns [21]. In this work, the authors propose transformations that implement most of the design patterns. Most of the efforts in this research direction concentrate on the definition of transformations and their implementation. To our knowledge, there is no effort on the automatic detection of the situations where this transformations can apply.

Several authors have addressed the particular problem of class hierarchy design and maintenance. In these work, 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 [10]. 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. Casais (1991) proposed a local reorganization algorithm for a class hierarchy that relies on the user to specify the immediate superclasses of a new class. Godin and Mili in [6] propose the use of concept (Galois) lattices and derived structures as of formal framework for dealing with class hierarchy design or reengineering that guarantees maximal factorization of common properties including polymorphism. The ARES algorithm builds the Galois subhierarchy while preserving initial *relevant* classes and also deals with the automatic detection of specialization relationships between properties [5]. The GURU tool proposed by Moore (in [14]) deals with refactoring of methods and the class hierarchy in an integrated manner. In [7], reengineered hierarchies are compared using global class hierarchy metrics.

3. Diagnosis of design flaws

Experienced designers/programmers have a relative precise idea on what should be a good application/program relatively to a quality perspective (maintainability, reliability, reusability, etc.) This knowledge is built from their experiences and from the common knowledge related to the design/programming paradigm. Books, like [19] for example, give a set of rules that help developing good and understandable programs. Most of the time, these rules cannot be implemented to detect automatically symptomatic situations in a design/code. The main reason is that these rules are by definition fuzzy. If we consider the rule that states that we have to avoid long methods or methods that contain a lot of variables, it is hard to derive a threshold for the size of a method or for the number of variables from which we consider that we have a symptomatic situation.

To solve this problem, two directions seem promising. The first one is to use fuzzy logic to implement the quality rules/models. The second direction consists of using these rules as starting hypotheses and deriving precise rules by the way of empirical studies (i.e. building quality estimation models). Due to lack of space, we focus in this paper on the second direction.

Symbol	Name
CLD	Class-to-Leaf Depth
NOC	Number Of Children
NMO	Number of Methods Overridden
NMI	Number of Methods Inherited
NMA	Number of Methods Added
SIX	Specialization Index
CBO	Coupling Between Object classes
DAC'	Data Abstraction Coupling
IH-ICP	Information-flow-based inheritance coupling
OCAIC	Others Class-Attribute Import Coupling
DMMEC	Descendants Method-Method Export Coupling
OMMEC	Others Method-Method Export Coupling

Table 1. Inheritance and coupling metrics

Roughly speaking, building a quality estimation model consists of establishing a relation of cause and effect between two types of software characteristics: 1) internal

attributes which are directly measurable such as size, inheritance and coupling, and 2) quality characteristics which are measurable after a certain time of use such as maintainability, reliability and reusability. The process we follow to build such models is based on classical machine learning algorithms, particularly C4.5 [17]. More details on the different steps can be found in [13].

Before giving an example on the obtained rules, we present in Table 1, the sets of metrics which will be used in this paper. Readers who are interested in the formal definitions of these metrics can find more details in [8].

The two following rules are examples quality estimation rules:

Rule INH7: *NMI(c) > 22 ⇒ class(c) = 1 [75.8%]*
Rule CPL1: *CBO(c) > 14 ⇒ class(c) = 1 [88.2%]*

Rule INH7 (for inheritance rule number 7), for example, states that a class c, which inherits more than 22 methods, is hard to maintain (level 1 of maintainability). This rule is valid for 75.8% of the learning examples (classes). Rule CPL1 (for coupling rule number 1) states that classes that uses or is used by more than 14 classes is hard to maintain. This rule is valid for 88.2% of the learning examples.

Beyond the fact that the magic numbers 22 and 14 depend on the sample used in the learning process, these rules show big values for CBO (Coupling Between Objects) and NMI (Number of Methods Inherited) are bad for design. They have also the merit to propose thresholds values that enable detecting design flaws.

4. Prescription

Once a symptomatic situation is detected using a quality model, the next step is to propose possible transformations that improve the quality of a program while preserving its behavior. Using the quality models, we can establish a cause-to-effect relationship between some combinations of metric values and a poor design quality. The principle of prescription is to define another relationship between transformations and the improvement of the quality. To derive such a relationship, we us the intuitive hypothesis which states that if a good design corresponds to a good combination of metric values, then there are strong chances that a good combination of metric values corresponds to a good design. For example, if NMI of a class c is less than 22 we can presume that c is not hard to maintain, with the hypothesis that no other negative rule applies.

This type of reasoning is close to what some researchers call the abductive inference. Usually, this type of reasoning is not highly reliable. However, in our case, it can give good results for the two following reasons:

- Our goal is to help the programmer/maintainer to concentrate on certain parts of the system which are possibly problematic and not to decide which transformations must be applied.
- The estimation models use metrics that measure user meaningful artifacts rather that derived metrics. This helps the programmer/maintainer to decide which prescription makes sense.

Up to now, we showed that by changing the values of certain metrics, we presume that we can improve the quality of an application/program. The problem to solve now is then, how to change the value of a metric? An intuitive solution is to find out which transformation (or set of transformations) allows changing the value of a particular metric (or set of metrics). For example, if the rule INH7 applies to a class d (NMI(d) = 24), which transformations allow decreasing the number of inherited methods while preserving the behavior of d?

To respond to such kind of question, we need to study the impact of a predefined set of transformations on a predefined set of metrics. The example of Figure 1 shows how we study the impact of the creation of an abstract class C_b to factorize a set of classes C_1, C_2, …, C_n on the inheritance metrics (see section 5.1 for the details). The creation process is composed of three steps (1) creating an empty class C_b in the same level as the classes C_1, C_2, …, C_n, (2) changing the superclass of these latest classes to C_b, and (3) abstracting the common methods and moving common code segments to C_b. We can notice that these three elementary transformations do not change the behavior of the involved classes.

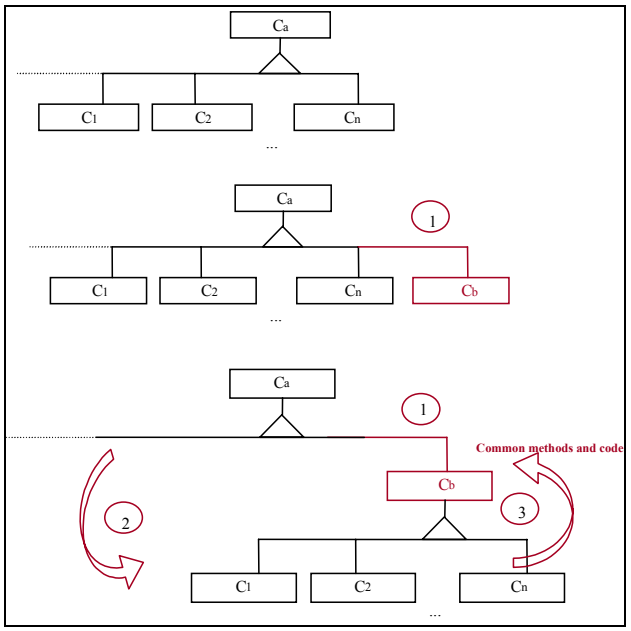

Figure 1. The three steps for the creation of an abstract class

To study the impact of the global transformations on the metrics, we first study the impact of each elementary transformation and then derive the global impact. Table 2 shows the impact of the elementary and global transformations on the inheritance metrics (defined in Table 1) of the different classes involved in the transformation process. This impact is given in the form of positive or negative change in the value of a metric. N is the number of factorized classes, M_A the number of abstracted methods, and M_C the number of methods created from the common code abstracting the common methods and moving common code segments.

	C_a	Anc. of C_a	C_i
(1) Creating C_b	NOC +1		
(2) Changing super-class of C_is	NOC -N CLD +{0, 1}	CLD +{0, 1}	
(3) Moving common methods and code from C_is to C_b			NMA - [0, $\|M_A\|$] NMO +[0, $\|M_A\|$] NMI +$\|M_C\|$
Global variations	NOC +1-N CLD +{0, 1}	CLD +{0, 1}	NMA - [0, $\|M_A\|$] NMO +[0, $\|M_A\|$] NMI +$\|M_C\|$

Table 2. Impact of the transformation of Figure 1 on the inheritance metrics

If we go back to the example of the class d, we can say that at least the transformation of Figure 1 cannot decrease the value of NMI and therefore it is not useful for this case.

5. A case study (Coupling/inheritance vs. maintainability)

In this section, we study the particular case of the diagnosis of bad maintainability by using the values of metrics for coupling and inheritance as symptoms. We particularly focus on the quality models used (and their corresponding metrics), on how to build a diagnosis and prescription tool, and on how to use this tool. We end this section by presenting and discussing some examples.

5.1 Quality models and derived metrics

The two models used in our study allow detecting the fault-prone classes (an important factor for the maintainability) using the values of respectively inheritance and coupling metrics. These models are trees in the beginning which are then transformed into rule sets using tree pruning. They were obtained by application of C4.5, a machine learning algorithm on a set of approximately 100 classes for which the number of faults is known. A classification 1 means that the class generated at least one fault. For a simplification purpose, we present here only the negative rules of each model (i.e. rules that give classification 1):

- Predicting the fault-proneness using inheritance metrics
  - *Rule INH5:*
    $CLD(c) = 0 \wedge NMA(c) > 7 \wedge SIX(c) > 0.222222$
    $\Rightarrow class(c) = 1$ [91.2%]
  - *Rule INH6:*
    $NOC(c) \leq 1 \wedge NMO(c) = 0 \wedge NMI(c) \leq 6$
    $\Rightarrow class(c) = 1$ [79.9%]
  - *Rule INH7:*
    $NMI(c) > 22$
    $\Rightarrow class(c) = 1$ [75.8%]
- Predicting the fault-proneness using coupling metrics
  - *Rule CPL1:*
    $CBO(c) > 14$
    $\Rightarrow class(c) = 1$ [88.2%]
  - *Rule CPL2:*
    $IH\text{-}ICP(c) > 16$
    $\Rightarrow class(c) = 1$ [87.1%]
  - *Rule CPL3:*
    $DAC'(c) \leq 2 \wedge OCAIC(c) > 0 \wedge OMMEC(c) > 9$
    $\Rightarrow class(c) = 1$ [83.3%]
  - *Rule CPL4:*
    $OCAIC(c) = 0 \wedge DMMEC(c) = 0$
    $\Rightarrow class(c) = 1$ [81.9%]

The metrics used in these models are given in Table 1 (only those that appear in the negative rules).

5.2 Building a diagnosis and prescription tool

The diagnosis part of the tool consists of an engine that applies the rules of the quality estimation models to the classes of a given system. We suppose that the values of the metrics were extracted beforehand. In our case, we use a separate tool for the extraction. We developed a generic tool called OO1 that can be extended automatically to support any quality model expressed in term of classification rules.

The prescription part of the tool is based on the analysis of the impact of the transformations on the metrics as presented in section 4. In this study we used three different transformations: (1) creating an abstract class (c.f. section

4), (2) creating specialized subclasses, and (3) creating an aggregated class.

Creating an abstract class. This transformation is presented in section 4. To be complete, we show, in Table 3, its impact on the coupling metrics. Note that N.D. means that there is an impact but we cannot determine whether it is positive or negative. P_A is the number of parameters added when creating the new methods from the common code.

Classes	Global variations
Classes that reference the abstracted methods	IH-ICP +i (I >= 0)
Classes referenced in the common code	CBO $-[0, N-1]$ DMMEC -i (i >= 0) OMMEC -i (i >= 0)
C_i	CBO N.D. IH-ICP N.D

Table 3. Impact of the transformation of Figure 1 on the coupling metrics.

Creating specialized subclasses. The aim of this transformation is to create new subclasses for a class that is initially a leaf of the inheritance tree. The candidate subclasses are determined from the detection of conditions that suggest new specialized abstractions. The class C_a is the initial class, the C_1, C_2,…, C_N classes are the created subclasses. C_a is assumed to initially have no descendant. The low-level transformations (steps) involved are:

- *Step 1*: Find conditional expressions for which conditions suggest subclasses.
- *Step 2*: For each condition create a subclass.
- *Step 3*: For each condition expression, create a method in each subclass. Simplify and specialize the method's body for each subclass according to the conditions represented by the subclass.
- *Step 4*: Specialize some or all of the expressions that create instances of the initial class.

Classes	Global variations
C_a	NOC +N CLD +1 NMA $+\lvert ExpCond \rvert$
Classes referenced in the code of conditional expressions	CBO $+[0, \lvert N \rvert]$ DMMEC +i (i >= 0) OMMEC +i (i >= 0)
Ancestors of C_a	NOD +N CLD +{0,1}

Table 4. Impact of creating specialized subclasses on the inheritance and coupling metrics

Using the same impact analysis technique as for the first transformation, the global metric variations for the classes impacted by this high-level transformation are summarized in Table 4. *ExpCond* is the set of conditional expressions, and N the number of created subclasses.

Creating an aggregated class. This transformation consists of grouping a subset of a class C_a members into a new class C_b. An instance of C_b will be part of an instance of C_a. We assume that other classes do not inherit the grouped members. The elementary transformations involved are:

- *Step1:* Create C_b and move the considered members.
- *Step2:* Insert a new attribute in C_a that will contain the instance of C_b.
- *Step3:* Modify the references to the transferred methods
- *Step4:* Delete the transferred members from C_a.

Table 5 summarizes the changes in the values of coupling and inheritance metrics. ATR is the set of transferred attributes. Note that no inheritance metric is affected by the transformation.

Classes	Global variations
C_a	CBO +1-i (i >= 0) DAC' $+1-[0, \lvert A\,\mathrm{TR} \rvert]$ OCAIC $+1-[0, \lvert A\,\mathrm{TR} \rvert]$ IH-ICP -i (i >= 0)
Classes referenced by the transferred methods	CBO +{0,1}

Table 5. Impact of creating an aggregated class on the inheritance and coupling metrics

5.3 Applying the corrector

As presented in the section above, the three transformations can vary the ranges of values for the metrics of the involved classes. This is what we precisely need to improve the quality of an application (see section 4). From the corresponding tables (Table 3 to Table 5), we can detect what are the transformations that can make the metrics values of a class fit into the desired range (good combination of metric values). Each column of a table is dedicated to one class or one category of classes involved in a transformation, thus choosing a particular column of a particular table determines both the transformation to apply and the role played by the class within the transformation context.

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. OO1 proposes all the possible transformations that can be applied when it detects a symptomatic situation. The user can then select the appropriate transformation.

The following algorithm gives a summary of the diagnosis and prescription as implemented by OO1

```
AC is the set of the classes of the application
For each class c of AC do
 - Calculate the metric values
 - Apply quality rules
 If a negative rule applies then
    - Choose the metrics and the desired changes to their values
    - Select transformations that allow these changes
        - Propose the transformations that correspond to the context of c
 EndIf
EndFor
```

5.4 Examples and discussion

To illustrate the approach proposed in this paper, we present in this section two examples of the application of OO1. These examples are classes of a multiagent system coded in C++ called LALO.

Example 1: The case of XrulesKB classes. Three classes were detected by OO1 as a bad design from the maintainability point of view according to rules INH6 and CPL4. These three classes are called respectively ExecRulesKB, OrdRulesKB and MsgRulesKB. The values for the inheritance and coupling metrics are given in Table 6.

Metrics	ExecRulesKB	MsgRulesKB	OrdRulesKB
CLD	0	0	0
NOC	0	0	0
NMO	0	0	0
NMI	0	0	0
NMA	9	9	9
SIX	0	0	0
CBO	3	3	3
IH-ICP	0	0	0
DAC'	0	0	0
ACAIC	0	0	0
DMMEC	0	0	0
OMMEC	2	2	2

Table 6. Inheritance and coupling metrics for the XRulesKB classes

To avoid that rule INH6 applies for each of the three classes, we have to increase the value of *NOC* or increase the value of *NMO* or increase the value of *NMI*. From Table 2, column C_i, we can suggest to create an abstract class for the three classes. As the three classes have 5 common methods (*add*, *remove*, *export_engine_data*, *registration* and the = operator), the NMO values for the three classes increase to 5, which is sufficient to avoid the application of rule INH6.

This prescription is appropriate according to the context of the application. We were not surprised to find in the same system, three other classes named respectively ExecRule, OrdRule and MsgRule with an abstract class Rule (see Figure 2).

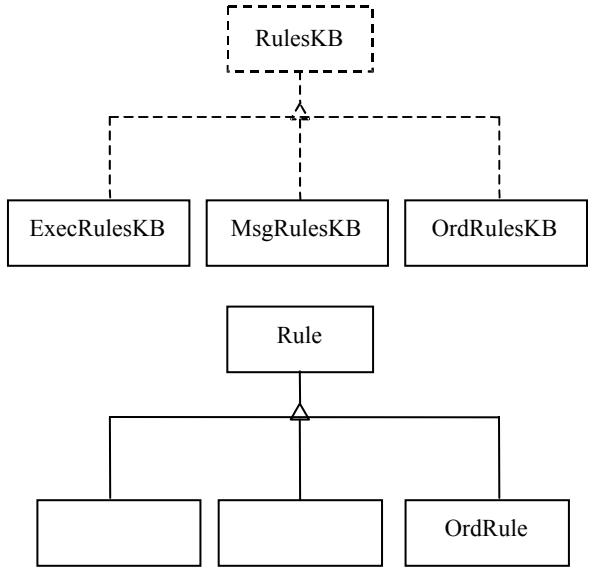

Figure 2. A partial view of the system LALO and the prescription of OO1 (dotted lines)

Another prescription is given by Table 4, column C_a. OO1 proposes to create for each class a set of specialized subclasses. The three classes are small and are already pretty much specialized. A user can then reject this suggestion. Figure 3 shows the suggestions of OO1 for this particular case.

From the coupling point of view, if we want to avoid the application of rule CPL4, we have to increase OCAIC or DMMEC at least by 1 (see rule CPL4 in section 5.1 and Table 1).

Increasing the value of OCAIC by 1 can be possible if the following conditions are true:

1. We can create an aggregated class from an XrulesKB class as stated by Table 5, column C_a (OCAIC $+1-\left[0,\left|A\,\mathrm{TR}\right|\right]$).

2. No attribute is transferred to the created class attributes (|*A*TR|=0).

After examining the content of the three classes, such a transformation is not concretely feasible. Therefore, a user can reject this prescription.

Another possible prescription is to create specialized subclasses to increase the value of *DMMEC* (see Table 4, column "Classes referenced in the code of conditional expressions"). Even if this transformation is theoretically possible, the particular context of the application does not allow it.

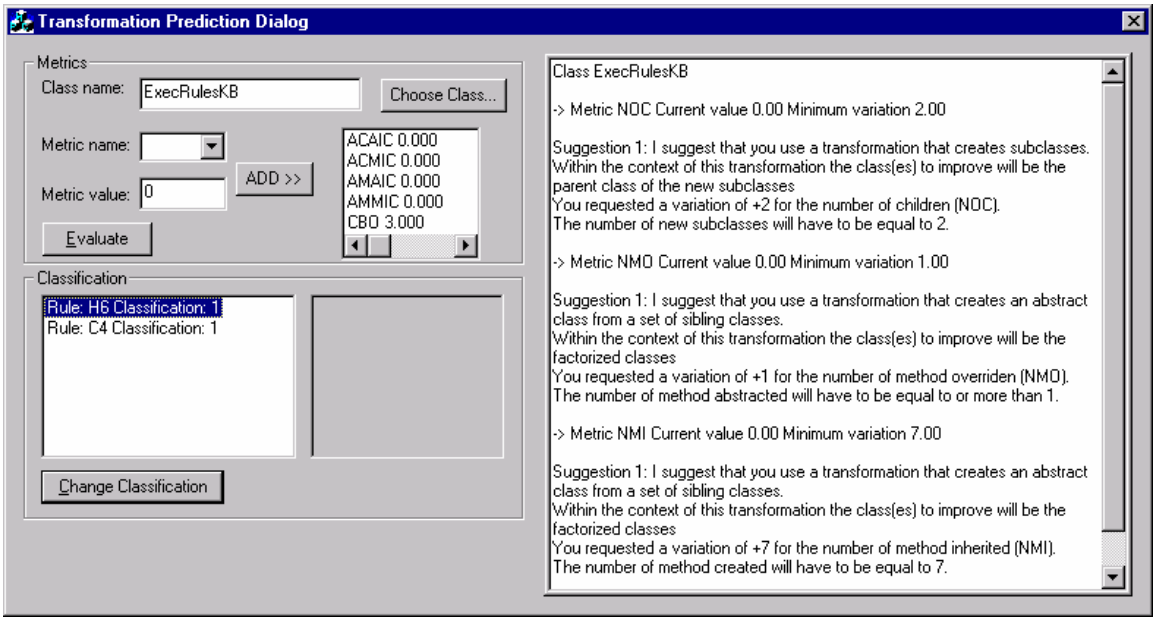

Figure 3. Some prescription alternatives given by OO1 for the case of XRulesKB classes

Example 2: The case of KQMLObject class. A class named KQMLObject was detected by OO1 as having bad maintainability according to rules INH5, CPL3 and CPL4 (see Table 7 for the values of metrics).

Metrics	**KQMLObject**
CLD	0
NOC	0
NMO	10
NMI	0
NMA	14
SIX	0.41
CBO	3
IH-ICP	0
DAC'	0
ACAIC	0
DMMEC	0
OMMEC	76

Table 7. Inheritance and coupling metrics for KQMLObject class

To prevent the application of rule INH5, we have to increase the value of CLD or decrease the value of NMA. Four possible transformations to increase CLD appear in the impact tables:

1. Create an abstract class as a direct subclass of KQMLObject (Table 2, column C_a).
2. Create an abstract class as descendent but non-direct subclass of KQMLObject (Table 2, column "Ancestors of C_a").
3. Create specialized subclasses of KQMLObject (Table 4, column C_a)
4. Create specialized subclasses of one of the subclasses of KQMLObject (Table 4, column "Ancestors of C_a").

As KQMLObject is a leaf class, only transformation 3 is proposed. All the others suppose that KQMLObject has subclasses. When we examined the code to verify if transformation 3 is concretely possible, we found that some condition expressions concern the possible values of a particular attribute (*performative_number*). This is an indicator of a bad design. Creating subclasses that corresponds to the different values of this attribute will simplify the code of KQMLObject.

Another possible suggestion was to propose to factorize KQMLObject with other possible classes to decrease the value of NMA according to Table 2 and column C_i. However, KQMLObject is already factored with the only sister it has (see Figure 4).

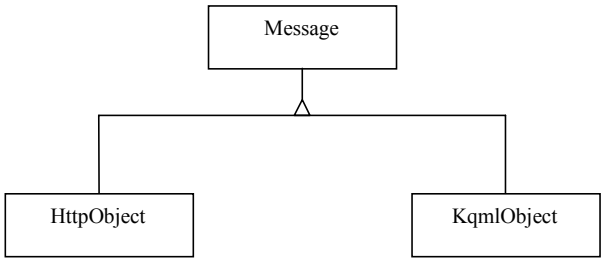

Figure 4. Hierarchy of KQMLObject

From the coupling point of view, the only interesting prescription is to create an aggregated class. This avoids the application of rules CPL3 and CPL4 by varying the value of OCAIC (Table 5, column C_a). The class KQMLObject is too large and too complex to make such a transformation possible. The only condition is to find a subset of methods to transfer without transferring any attribute ($+1-\left[0,\left|A\,\mathrm{TR}\right|\right] > 0$).

6. Conclusion

In this work, we have investigated the use of metrics for detecting potential design flaws and for suggesting potentially useful transformations for correcting them. Initial experiments with the OO1 prototype have demonstrated the feasibility of the approach and its usefulness. Indeed, our approach can help a designer/programmer by suggesting transformations. It can help her or him also focusing on a particular part of a large system.

From the perspective of automation, the response to the paper title question can be yes and no. Yes, using metrics is a step towards the automation of quality improvement. If we look to the whole process of detecting flaws and correcting them, metrics can help automating a large part of it. The response can be no. Indeed, the results of our experiments show that a prescription cannot be executed without a validation of a designer/programmer. Our approach cannot capture all the context of an application to allow such a type of automation.

A direction that we will explore in our future work is to better capture the context of an application. This will enable us to refine the suggestions by eliminating those that are not relevant. Another direction is to study the adaptation of our approach to suggest the use of design patterns. As mentioned in section 2, Tokuda and Batory show that it is possible to automate the implementation of a design pattern into an existing code. However, this approach cannot automate why and where implementing the design pattern.

Acknowledgment

The authors would like to thank Pr. Hafedh Mili for his comments on this work.

References

[1] Basili V., Briand L. & Melo W., How Reuse Influences Productivity in Object-Oriented Systems. *Communications of the ACM,* Vol. 30, N. 10, pp104-114, 1996.

[2] Casais E., *Managing Evolution in Objet Oriented Environments: An Algorithmic Approach*, thèse de Doctorat, université de Genève, 1989.

[3] Chidamber S. & Kemerer C. A Metrics Suite for Object-Oriented Design, *IEEE Transactions on Software Engineering*, June, 1994, p. 476-492.

[4] Demeyer S., Ducasse S., Metrics, *Do they really help ?*, In Proc. of LMO, 1999.

[5] Dicky, H., Dony, C., Huchard, M. & Libourel, T. On Automatic Class Insertion with Overloading. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*, CA, USA: ACM SIGPLAN Notices, pp. 251-267, 1996

[6] Godin, R. & Mili, H. Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, A. Paepcke (Ed.), Washington, DC: ACM Press, pp. 394-410, 1993.

[7] Godin, R., Mili, H., Mineau, G. W., Missaoui, R., Arfi, A. & Chau, T.-T. Design of Class Hierarchies based on Concept (Galois) Lattices. *Theory and Application of Object Systems*, **4**(2), 117-134, 1998

[8] Ikonomovski, S., *Detection of Faulty Components in Object-Oriented Systems using Design Metrics and a Machine Learning Algorithm*, Master Thesis, Mc Gill University, Montréal, 1998.

[9] Li W. & Henry S., Object Oriented Metrics that Predict Maintainability. *Journal of Systems and Software*. Vol.23, No.2., 1993.

[10] Lieberherr, K. J., Bergstein, P. & Silva-Lepe, I. From Objects to Classes: Algorithms for Optimal Object-Oriented Design. *Journal of Software Engineering*, **6**(4), 205-228, 1991.

[11] Lounis H., Melo W., Sahraoui H. A*., Identifying and Measuring Coupling in OO systems*, technical report CRIM-97/11-82, 1997.

[12] Lounis H., Sahraoui H. A., Melo H. A., Towards a Quality Predictive Model for Object -Oriented Software, *L'Objet*, Volume 4 (4), Ed. Hermes. 1998 (in french).

[13] Mao Y., Sahraoui H. A. and Lounis H., Reusability Hypothesis Verification Using Machine Learning Techniques: A Case Study, *Proc. of IEEE Automated Software Engineering Conference*, 1998.

[14] Moore, I. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*, CA, USA: ACM SIGPLAN Notices, pp. 235-250, 1996

[15] Opdyke F. W., *Refactoring Object-Oriented Frameworks*, PhD thesis, University of Illinois, 1992.

[16] Opdyke F. W. & Johnson E. R., Creating Abstract Superclasses by Refactoring, in *Proceeding of CSC'93: The ACM 1993 Computer Science Conference*, February 1993.

[17] Quinlan J. R., *C4.5: Programs for Machine Learning,* Morgan Kaufmann Publishers, 1993.

[18] Roberts D., Brant J., Johnson E. J.: A Refactoring Tool for Smalltalk*, Theory And Practice of Object Systems*, Volume 3 (4): 253-263, (1997).

[19] S. Skublics, E. J. Klimas, D. A. Thomas, *Smalltalk with Style*, Prentice Hall, 1996.

[20] Sommerville I., *Software Engineering*, Addison Wesley, fourth edition, 1992.

[21] L. Tokuda and D. Batory, Evolving Object-Oriented Designs with Refactorings, *Proc. of IEEE Automated Software Engineering Conference*, 1999.