Maintainability Metrics of UML Design

Iryna Zaretska[1] and Maryna Besedina[1]

[1] V.N. Karazin Kharkiv National University, Kharkiv, Ukraine
zar@univer.kharkov.ua, reencka@rambler.ru

Abstract. The paper introduces a new object-oriented metric to evaluate maintainability of the software system at the design stage. Unlike well known object-oriented metrics applicable only to one class or to a category of several interconnected classes the proposed one evaluates the degree of extensibility for the whole static design. The metric is based on the main principles of object-oriented design and can be used by designers for evaluation and refining purposes. The results of the experiment on real projects to check this metric are reported. The Java plug-in for calculating this metric in UML Case tools is presented.

Keywords: Software design, object-oriented approach, software maintainability, object-oriented metrics.

Key Terms: SoftwareSystem, SoftwareComponent, Object, Model, Metric.

1 Introduction

The importance of evaluating the quality characteristics of the software under development at the early stages of the software lifecycle is well known in software engineering. One of such characteristics much valued at present times is its maintainability, which according to the ISO / IEC 9126 standard means “the capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.” [1]. So the main task of a designer of an object oriented software system is to produce the design easily adaptable to new or changed requirements without considerable or even any changes in the already existing coded and tested implementation. How to calculate the degree of the future system’s maintainability by the UML design?

There exist a lot of object oriented metrics to evaluate the quality of the UML design but they are limited to evaluating its complexity, or using object oriented paradigm or reusability of a class or a group of coupled classes. Neither of these metrics can tell how much effort will be needed to enhance the design in order to adapt it to new requirements. Here we restrict such enhancement to using inheritance and polymorphism so that changes in the requirements can be satisfied only by creating new classes derived from already present in the design and overriding their methods or by creating new implementations of the already present in the design interfaces.

We introduce a metric, we call it E (from "Enhancement"), which is calculated by a class diagram and shows how easily new classes can be added to the design. In fact it uses well known in object oriented development main principle which recommends "programming to interfaces, not to implementation" [2]. It is up to the designer to decide how to use its value and whether to reconsider the design taking into account the application domain and possible future enhancements of the system.

2 Related Work

There exist a lot of object oriented metrics and sets of metrics to evaluate the quality of software design. They are divided into categories and used for different purposes. For example, Java Eclipse provides quite a big number of such metrics calculated by the project's code. Despite the variety of these metrics the question of their values interpretation remains open. Usually it is more qualitative then quantitative interpretation like "the less the better" or "the closer to 1 the better". It always depends on the application domain and the scope of the project how close to 0 or to 1 is good enough. It is even harder to interpret the metrics with absolute values such as DIT (Depth of Inheritance Tree) or NOC (Number Of Children). Several case studies [5, 6] report finding the most valued metrics and their weighted integral characteristic to estimate some special quality characteristics of the software system.

The closest metrics to our studies are the following ones [3, 4]. The instability metric $I = Ce / (Ca+Ce)$, where Ca - stands for a number of Afferent Couplings, Ce - stands for a number of Efferent Couplings. This metric evaluates the instability of a category of classes: $I=0$ means absolutely stable (reusable) category while $I=1$ means impossibility to reuse the category as a whole. The abstractness metric $A = nA / nAll$, where nA is a number of abstract classes in a category and $nAll$ is an overall number of classes in this category also is applicable to the category of classes. The equality $A=1$ means that the category is completely abstract and should be enhanced by inheritance to be used in a "live" software while $A=0$ shows a completely concrete category which is not good for enhancement purposes. In fact these two metrics work together forming the so called main sequence – a straight line given by the equation $A+I=1$ (Fig.1). This line defines the categories with the best balance between abstractness and instability.

Two more metrics calculated as the distance from this line $D=|(A+I-1)/sqrt(2)|$ and the normalized distance from this line $Dn=|A+I-2|$ are also used to evaluate the categories of classes. These metrics could be used to some extent to evaluate the degree of maintainability of the category but not of the whole system. Besides some efforts are needed to automatically allocate categories and calculate these metrics.

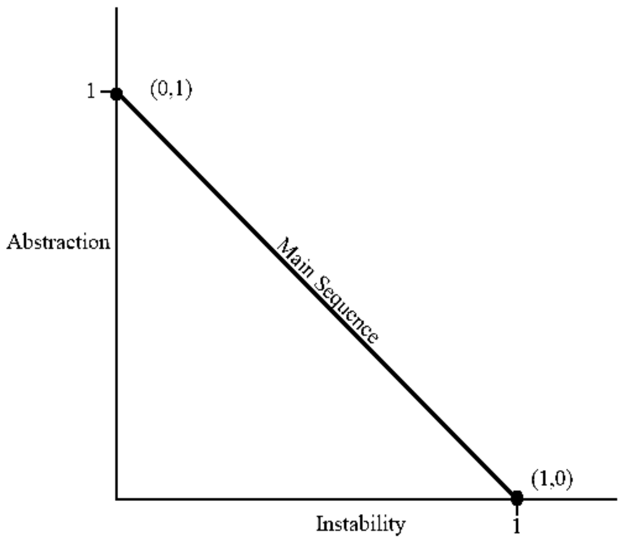


Fig. 1. Main sequence.

3 Maintainability Metrics

To create really object oriented design one has to follow several simple principles [2]:

- use interfaces to define common responsibilities of classes;
- declare variables to be instances of the interfaces, not instances of particular classes;
- use creational patterns to associate interfaces with implementations;
- keep classes focused on one task;
- if some responsibility of a class could change in future, create a separate interface to declare this responsibility and its present implementation, and use delegation technique.

Our maintainability metric is based exactly on these principles. We calculate the overall number of connections between classes on the class diagram and denote it by *nC*, then we distinguish those from them which connect a class to an interface or an interface to an interface, their number we denote by *nI*. Their ratio shows the degree to which the principles mentioned above are satisfied.
So we introduce the metrics calculated on the class diagram by the formula $E = nI / nC$, where *nI* is a number of the”class - interface” or “interface - interface” connections and *nC* is the overall number of connections. If $nI=nC=0$ (no connections at all) we put $E=0$.

The connection of the “class – interface” type can be as follows:

— association between a class and an interface (usually means composition);,
— implementation of an interface by a class;
— dependency between a class and an interface (usually means local visibility: a parameter of a method or the return value or the local variable of the method);
— aggregation of an interface in a class (the "whole-part" relations, containers, etc.)

As usual for such metrics its value is between 0 and 1: the closer to 1 the better. Let us see how it works in a simple well known example with validation of data. If we put the responsibility of the data validation onto a class (Fig. 2) any changes in validation rules will require changes in this class. Here *E=0*.

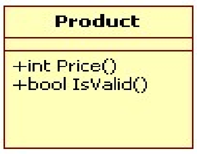


Fig. 2. Class Product is responsible for data validation.

But if we consider the main principles mentioned above the design will look otherwise (Fig.3).

Now *E=(2+1)/3=3/3=1* which is the best value. New rules of validation will require new implementations of the base interface without any changes in the already existing design.

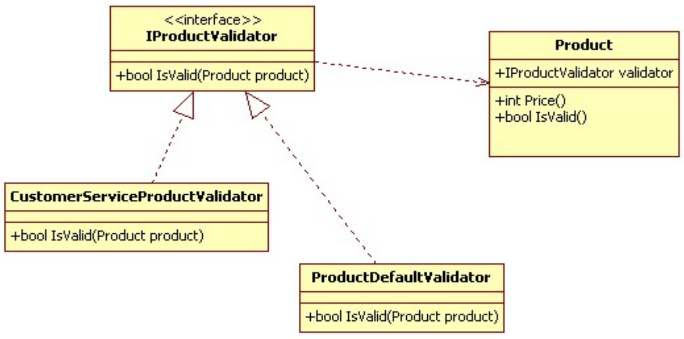

<<interface>>
IProductValidator
+bool IsValid(Product product)
Product
+IProductValidator validator
+int Price()
+bool IsValid()
CustomerServiceProductValidator
+bool IsValid(Product product)
ProductDefaultValidator
+bool IsValid(Product product)

Fig. 3. Responsibility for data validation is delegated to another class.

To be certain we conducted an experiment with five middle sized projects of different scope but all developed inside University. We engaged experts to assess their quality by the following scale: 1 – needs considerable redesigning to enhance; 2 –needs small changes in existing design to enhance; 3 – no need to change existing design to enhance. Then we calculated *E* metric. The results are given in a Table 1.

The resulting value of *E* metric is useful for the designer just to ask himself if all the principles of object oriented design satisfied and if not to find reasonable grounding in the concrete application domain or/and requirements.

Table 1. The results of experiment.

Name of the project	Overall number of classes in the design model	Overall number of interfaces in the design model	Expert evaluation	Value of *E* metrics
Bug Tracker	20	3	1	0,143
Tester	71	7	2	0,527
Timetable	11	0	1	0
Dean's office	43	5	2	0,51
Preparatory Department	46	17	3	0,64

4 Java Plug-in for Calculating *E* Metrics

To make the calculation of the *E* metric for big projects simple we developed the Java plug-in, we called it EParser, which can be easily added to many UML CASE tools with open source. We tried it with Eclipse. The main idea is to parse the XMI file of the class diagram generated by a UML CASE tool, find there elements (in fact connections) we are interested in and make needed calculations. The class diagram of the EParser is given in Fig. 4. A simple window shows the result (Fig. 5).

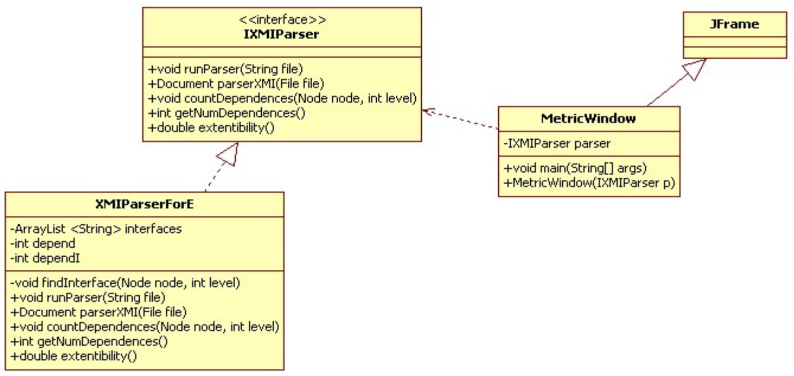
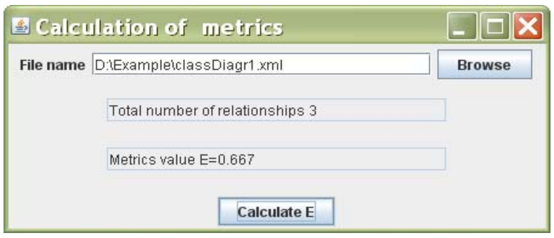

<<interface>>
IXMIParser
+void runParser(String file)
+Document parserXMI(File file)
+void countDependences(Node node, int level)
+int getNumDependences()
+double extentibility()
JFrame
MetricWindow
-IXMIParser parser
+void main(String[] args)
+MetricWindow(IXMIParser p)
XMIParserForE
-ArrayList <String> interfaces
-int depend
-int dependI
-void findInterface(Node node, int level)
+void runParser(String file)
+Document parserXMI(File file)
+void countDependences(Node node, int level)
+int getNumDependences()
+double extentibility()

Fig. 4. Class diagram for EParser.



D:\Example\classDiagr1.xml

Total number of relationships 3

Metrics value E=0.667

Fig. 5. Snapshot of working plug-in.

5 Conclusions

We introduced the metric that shows the degree to which the main principles of object oriented design are satisfied. As the object-oriented style provides for flexibility, reusability and maintainability of the software, this metric serves the same purpose. Certainly it is only partial indicator as it is calculated only by static decomposition of the system and does not consider its dynamic aspects. However it can be useful reminder to the designer to think about possible future changes of the system and check if the design is ready to adopt them easily and naturally.

References

1. ISO/IEC TR 9126-3:2003 Software engineering — Product quality — Part 3: Internal metrics (2003)
2. E.Gamma, R. Helm, R. Johnson, J. Vlissides: Design patterns: Elements of reusable object oriented software. Addison-Wesley (2001)
3. R.C. Martin: Designing Object-Oriented C++ Applications Using the Booch Method. Prentice-Hall (1995)
4. R. C. Martin: OO Design Quality Metrics. An Analysis of Dependencies (1994)
5. L. C. Briand, J. Wust, H. Lounis: Replicated case studies for investigating quality factors in object-oriented design. . Empirical Software Engineering. Vol. 6, Issue 1, Springer Netherlands, 11--58 (2001), http://www.scopus.com/
6. L. C. Briand, J. Wust, J. Daly, D. V. Porter: Exploring the relationship between design measures and software quality in object-oriented systems. Journal of Systems and Software, Volume 51 Issue 3, Elsevier Science Publishing Company, Inc., 245--273 (2000)