

A Model of Object-Oriented Design to Support the Definition of Metrics

Ralf Reißing

Abt. Software Engineering, Institut für Informatik, Universität Stuttgart
Breitwiesenstr. 20-22
D-70565 Stuttgart
reissing@informatik.uni-stuttgart.de

Abstract: Object-oriented design has a pivotal role in software development because it determines the structure of the software solution. Once the design has been implemented, it is difficult and expensive to change. Therefore the design should be good from the start. Metrics can help to evaluate and improve the quality of a design. Many metrics of object-oriented design have been proposed. Unfortunately, most of these metrics lack a precise and unambiguous definition. However, such precise definitions are needed to enable automation of design evaluation. Therefore the definitions should be based on a formal model of design.

In this paper a formal model for object-oriented design called ODEM (Object-oriented DEsign Model) is presented including extensions to an earlier paper on ODEM. ODEM serves as a foundation for the formal definition of object-oriented design metrics. The model is based on the UML meta-model, that provides a formal model of object-oriented designs expressed in UML, the most widespread design notation. Examples show how object-oriented design metrics can be defined using ODEM.

1. Introduction

Design is an important cost driver in software development. The characteristics of the design, e.g. flexibility, heavily influence the ease of implementation and maintenance. As at least 50% of the total life-cycle cost goes into maintenance [Bo76], it is very cost effective to have a good design early and to maintain high design quality throughout the life-cycle. If bad design is not fixed in the design phase, the cost for fixing it after software delivery is between 5 and 100 times higher [BB01].

In order to create and maintain a high quality design, quality assurance in the form of design evaluation and review is needed. It is important and useful to measure design quality early in software development [CG90][Ro90]. Strong correlations between design metrics (e.g. modularity metrics) and the maintainability of systems have been identified. So design measurement is useful for quality assurance because it can uncover design problems that lead to bad maintainability. Early design measurement means, of course, that it should not depend on detailed design information or even on code. For quality assurance, the design metrics have to be embedded in a quality model to make them really useful for design evaluation. The quality model determines which quality aspects are relevant, which aspect(s) a metric belongs to, which measurement values are acceptable, and which values hint to design problems.

Design evaluation can be done by experts using checklists or questionnaires which are based on the quality model and its metrics. However, it is more efficient to automate design evaluation as far as possible by using a tool, especially when large designs are evaluated. Even though a tool can never replace a human expert, it can help him (or her) to identify design components that are (potentially) troublesome. The tool can also help to compare design alternatives by evaluating each alternative and comparing the results. Implementing such a tool, however, requires a precise, formal definition of the metrics.

As discussed in [Re01a], most metrics proposed so far do not have such a formal definition. Therefore a formal design reference model is needed for precise metric definition. A first version of a formal model called ODEM was presented in [Re01a]. In this paper, extensions to ODEM are presented that make it easier to use. In section 2 the basic ideas of ODEM are introduced, and section 3 gives an overview of the basic model elements described in depth in [Re01a]. In section 4 additional model elements are presented. Examples for the use of ODEM for metric definition are given in section 5. Finally, as an outlook, possible extensions to the model, tool support, and its use in design evaluation are discussed.

2. ODEM – An Object-Oriented Design Model

The formal model of object-oriented design proposed above is now introduced. It is called ODEM, which is an acronym for Object-oriented DEsign Model. “Odem” is an ancient German word for breath. ODEM is a very suitable name for the formal model because having such a model is as vital for precise metric definition as breathing is for living.

ODEM is based on the UML meta-model (see [OMG00a]), namely the part of the meta-model needed to capture class diagram elements. Defining metrics using the UML meta-model directly is difficult due to its high complexity. Therefore additional abstractions in the form of sets, attributes, and relations were defined as a layer on top of the meta-model. These additions (described in section 3 and section 4) make metric definitions easier to read and understand. The resulting structure is depicted in figure 1.

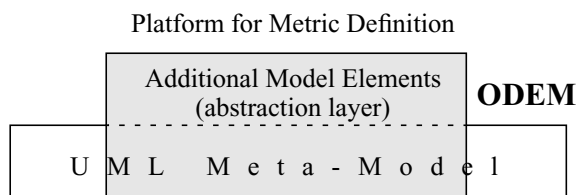


Figure 1: Conceptual structure of ODEM

Currently ODEM is restricted to the structural aspects of the UML (class and package diagrams) because these diagrams are most likely to be present and complete in design descriptions. In order to keep the model size manageable, the number of model elements and their attributes have been reduced to a minimum. For example, template classes and classes nested into classes are not featured. Also methods, i.e. implementations of opera-

tions, are not included. From the viewpoint of high-level design, the focus of ODEM, methods are just an implementation detail.

The UML specification also defines the Object Constraint Language (OCL, see [WK99]) as a part of the UML. The OCL is a formal language for expressing constraints on UML models. The author tried to use the OCL as a specification language for design metrics, but it did not seem to be suited for this purpose. Some metrics were extremely hard to express in OCL, and most of the resulting metric definitions were hard to read. Nevertheless, Abreu recently successfully used OCL for metric definition [Ab01]. However, instead of the UML meta-model he uses his GOODLY model as a reference model.

3. Basic Model Elements

Figure 4 shows an UML-like overview of the basic ODEM model elements and relations. The classes are taken from the UML meta-model, while the relations are derived from its relationship classes and compositions. Table 1 gives an overview of the basic formal identifiers of sets and relationships in ODEM. Details for the definitions of each model element based on the UML metamodel can be found in [Re01a]. The attribute visibility has been added later to package, class, and interface. It captures visibility information of elements contained in packages.¹

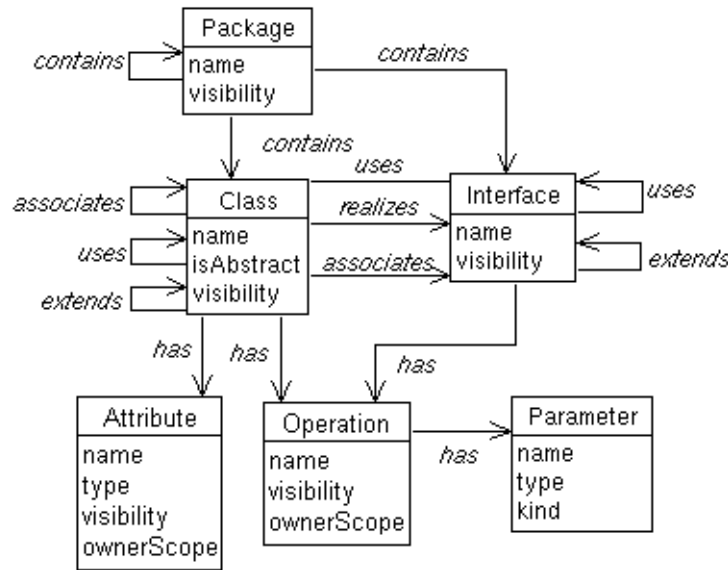


Figure 4: Overview of the basic ODEM elements

1. In the UML metamodel, this kind of visibility information is modeled by the association class ElementOwnership that belongs to the association between Namespace and ModelElement.

Identifier	Meaning
A	Set of all attributes in S
C	Set of all classes in S
I	Set of all interfaces in S
M	Set of all parameters in S
O	Set of all operations in S
P	Set of all packages in S (including S)
S	The system (contains all model elements)
associates	Relation for association relationships between classes/interfaces
contains	Relation for packages containing classes, interfaces, and packages in a package
extends	Relation for inheritance of features
has	Relation for classes/interfaces having features or operations having parameters
realizes	Relation for realization of an interface by a class
uses	Relation for use of a class/interface by a class/interface

Table 1: Formal identifiers in ODEM

4. Additional Model Elements

This section introduces further relationships and attributes derived from the basic relationships, which make the definition of metrics more convenient. Some of these relationships have also been outlined in [Re01a], but the definitions in this paper are more rigorous and complete. Additionally, necessary changes for covering template classes are described.

4.1 Derived Relations

The general dependency relation *depends_on* captures all kinds of relationships between classifiers, i.e. classes and interfaces: extension, realization, association, and usage. Note that it is assumed here that all kinds of usage (like use by operation call, attribute type, parameter type, or return type) were properly modeled in the design description by a dependency (which could be added automatically for the last three example cases if necessary).

$$depends_on(x,y) \text{ iff } extends(x,y) \vee realizes(x,y) \vee associates(x,y) \vee uses(x,y)$$

From this relation we can derive a relation for *package* dependencies. A package *p* depends on another package *q* iff a class/interface in *p* depends on a class/interface in *q*:

$$depends_on(p,q) \text{ iff } \exists c,c' \in C \cup I: contains(p,c) \wedge contains(q,c') \wedge depends_on(c,c')$$

4.2 Extended Relations

So far, the relations were defined without taking inheritance into account. To include inherited relationships, additional relations are introduced. First of all we need the transitive closure *extends** of the *extends* relation:

$$\text{extends}^*(x,y) \text{ iff } \text{extends}(x,y) \vee \exists z \in C \cup I: (\text{extends}(x,z) \wedge \text{extends}^*(z,y)).$$

Now the other extended relations (*associates**, *depends_on**, *has**, *realizes**, and *uses**) can be defined based on *extended**. As an example, the definition of *uses** is given. The other extended relations can be defined analogously.

$$\text{uses}^*(x,y) \text{ iff } \text{uses}(x,y) \vee \exists z \in C \cup I: (\text{extends}^*(x,z) \wedge \text{uses}(z,y))$$

Finally, an extended version of the contains relation is introduced which is the transitive closure of contains:

$$\text{contains}^*(x,y) \text{ iff } \text{contains}(x,y) \vee \exists z \in C \cup I: (\text{contains}(x,z) \wedge \text{contains}^*(z,y)).$$

Of course, because of the hierarchical package structure *contains**(S,x) holds for all elements x of P, C, and I (except for S itself).

4.3 Weights for Relations

The relations defined so far do not distinguish how many relationships of a kind exist between two model elements. It may make a difference, however, if a class has, say, one or three associations with another class. Therefore an additional attribute *weight* is introduced. The weight of each *contains*, *extends*, *has*, and *realizes* relation is 1. For the *associates*- and the *uses*-relation it is possible that there is more than one relationship of a kind between two model elements, therefore the weight is the number of these relationships. The *depends_on*-relation for classifiers as an aggregate relation has a weight that is the sum of the weights of the relations included

$$\begin{aligned} \text{depends_on}(x,y).\text{weight} &= \text{extends}(x,y).\text{weight} + \text{realizes}(x,y).\text{weight} \\ &+ \text{associates}(x,y).\text{weight} + \text{uses}(x,y).\text{weight} \end{aligned}$$

Non-existing relations are assumed to have a weight of 0. For the package *depends_on* relation, the weight is the number of relationships between classifiers in the packages.

$$\begin{aligned} \text{depends_on}(p,q).\text{weight} &= \\ \sum_{c \in C \cup I: \text{contains}(p,c)} \sum_{c' \in C \cup I: \text{contains}(q,c')} \text{depends_on}(c,c').\text{weight} \end{aligned}$$

For the *-relations, additionally the weight of inherited relations has to be taken into account. *contains** always has a weight of 1. For *extends** the weight is the number of inheritance paths between the first and the second model element.

$$\text{extends}^*(x,y).\text{weight} = \text{extends}(x,y).\text{weight} + \sum_{z \in C \cup I: \text{extends}(x,z)} \text{extends}^*(z,y).\text{weight}$$

Finally, for *associates**, *depends_on**, *realizes**, and *uses** the weight is the sum of the weights of the relations between the first model element and all of its superclasses with the second model element. Here the definition is shown for *uses**. The weights for the other relations are calculated the same way.

$$uses^*(x,y).weight = uses(x,y).weight + \sum_{z \in C \cup I: extends^*(x,z)} uses(z,y).weight$$

4.4 Adding Template Classes

This section gives an overview of changes to ODEM that make it possible to cover template classes. The template concept of the UML is quite general. It does not only apply to classes but to all model elements, e.g. collaborations, so design patterns can be expressed as template collaborations. Here only template classes are considered. Template parameters can be classes, values, and operations. A template can define new attributes and operations using the template parameters. A template can have uni-directional outgoing relationships with other classes, and it can also have relationships (e.g. extends, associates) with its parameters if they are of a class type. Template parameters, attributes, and operations may not show up in the sets C, A, and O respectively, as they are only “dummy” placeholders for real model elements.

Template instances provide a complete binding for the formal parameters of a template, thus creating a new class. This class of course belongs to the set C, but the template itself does not. Therefore a new set T is introduced that contains the template classes. A relation *binds*: (C×T) connects the instance of a template with the template. Thus, template instances can be differentiated from “normal” classes by checking if they have a *binds* relation.

When a template is instantiated, the relationships to other classes and to template parameters as well as the attributes and operations are “instantiated”, too: the template is copied, then the formal parameters are substituted by the actual parameters from the binding, and the template itself is replaced by the template instance. The attributes and operations of the template instance become members of the set A and O respectively, individually for each instance. New relations are created for each instance, too.

Templates are not part of the final, so-called „effective“ UML model, only their instances are. Therefore it is questionable if T and *binds* should really be part of ODEM from an UML point of view. On the other hand, the use of templates can reduce the complexity of a design, so for measuring design complexity it is worthwhile to preserve the original template information.

5. Example Metric Definitions

Using ODEM, design metrics can be defined. Here only a few examples taken from the set of metrics used by the quality model QOOD (see section 6.3) are given to demonstrate how the metric definition with ODEM works. There are three categories of metrics: class, package, and system metrics.

5.1 Class Metrics

The metric NAC (number of attributes of a class) counts all attributes of a class (including the inherited attributes):

$$NAC(c) = |\{a \in A: has^*(c,a)\}|$$

Refinements of this metric can count only public attributes, only class attributes, or only locally defined attributes:

NAC_1 (number of public attributes of a class)

$$NAC_1(c) = |\{a \in A: has^*(c,a) \wedge a.visibility = public\}|$$

NAC_c (number of class attributes of a class)

$$NAC_c(c) = |\{a \in A: has^*(c,a) \wedge a.ownerScope = classifier\}|$$

NAC_l (number local attributes of a class)

$$NAC_l(c) = |\{a \in A: has(c,a)\}|$$

When counting relationships, the weights of the relationships have to be considered. The metric NEDC (number of efferent dependencies of a class) counts all dependencies from a class to other classes or interfaces (the term “efferent” was adopted from [Ma95]):

$$NEDC(c) = \sum_{c' \in C \cup I} depends_on^*(c,c').weight$$

Refinements of this metric can discriminate between different dependency types (e.g. use), if the dependency was defined locally or inherited, or if dependency is on an abstract class or an interface.

NEUC (number of efferent uses relationships of a class)

$$NEUC(c) = \sum_{c' \in C \cup I} uses^*(c,c').weight$$

$NEDC_l$ (number of local efferent dependencies of a class)

$$NEDC_l(c) = \sum_{c' \in C \cup I} depends_on(c,c').weight$$

$NEDC_a$ (number of abstract efferent dependencies of a class)

$$NEDC_a(c) = \sum_{c' \in C \cup I: c'.isAbstract} depends_on^*(c,c').weight$$

5.2 Package Metrics

The metric NCP (number of classes in a package) counts the classes directly contained within a package p :

$$NCP(p) = |\{c \in C: contains(p,c)\}|$$

This metric can be refined with regard to the visibility of the classes (e.g. public), or if the classes are abstract.

NCP_1 (number of public classes in a package)

$$NCP_1(p) = |\{c \in C: contains(p,c) \wedge c.visibility = public\}|$$

NCP_a (number of abstract classes in a package)
 $NCP_a(p) = |\{c \in C: contains(p,c) \wedge c.isAbstract\}|$

As shown in section 4.1, package dependencies can be derived from class dependencies. The metric NEDP (number of efferent dependencies of a package) counts these dependencies of a package p on other packages:

$$NEDP(p) = \sum_{p' \in P \setminus \{p\}} depends_on^*(p,p').weight$$

This metric can be refined with regard to discarding inherited dependencies:

$NEDP_l$ (number of “local” efferent dependencies of a package)
 $NEDP_l(p) = \sum_{p' \in P \setminus \{p\}} depends_on(p,p').weight$

5.3 System Metrics

System metrics are mostly aggregate metrics, e.g. the metric NIS (number of interfaces in the system) that counts the interfaces in the system:

$$NIS(S) = |I|$$

A special case is the metric NAS (number of attributes in the system) that counts the attributes in the system, counting each attribute once regardless of inheritance.

$$NAS(S) = |A|$$

If the number of attributes of each class (NAC) were summed up, the result would possibly be higher due to inheritance. Instead, NAC_l should be used for summation. Like for NAC, refinements of NAS with regard to visibility or owner scope are possible.

6. Conclusion and Outlook

Early design evaluation is important for assuring software quality. Metrics can help in evaluating the design, but they have to be defined precisely. So their definitions should be based on a formal model of object-oriented design. In this paper, such a model called ODEM (based on the UML meta-model) was described and examples for its use for metric definition were given. The examples of metric definitions, together with the case studies in [Re01a] on existing metrics suites by Martin [Ma95] or Chidamber, and Kemerer [CK94] show that ODEM meets its purpose as a foundation for structural high-level design measurement. Its limits of applicability to detailed design measurement are due to the deliberate restrictions. As design measurement is best done early in development, it should not depend on detailed design information or on code.

6.1 Future Extensions

Future extensions of the formal model can incorporate the missing elements from the UML meta-model for the class diagram features, e.g. nested classes. To include dynamic

information, the full UML meta-model might be used for accessing the information from action, state, and interaction diagrams, no matter how incomplete they may be. An alternative would be to use code for the extraction of detailed design information. Code would be available in maintenance or in later iterations of an iterative development process.

6.2 Automation of Measurement

Because ODEM is based on the UML meta-model, implementing tools that calculate metrics defined with ODEM is not difficult. The implementation can be based on a tool that parses XMI² model descriptions generated by a UML tool. The measurement functionality could also be directly integrated in a UML tool if it supports scripting or plug-ins (e.g. Rational Rose, Together) or if it is open source (e.g. ArgoUML), but then it would be specific for the tool.

As most major UML tools by now are able to export XMI, the XMI-based measurement tool is superior because it is independent of the UML tool. The structure of ODEM makes it easy to store the model information in a relational database, so the metrics can be expressed as SQL statements. Currently a tool that extracts ODEM information from XMI files and stores it in a database has been implemented in a master's thesis project.

6.3 Use in Design Quality Assurance

Design measurement in itself, i.e. without a purpose, is useless because the actual measurements have no concrete interpretation. Therefore the metrics have to be associated with a quality model. The quality model determines the interpretation of the measurements and thus defines the notion of design quality. Currently the quality model QOOD (Quality of Object-Oriented Design, see [Re00][Re01b]) is developed by the author. QOOD uses ODEM as a foundation for the definition of its metrics set.

Another tool currently developed uses the ODEM database described above to calculate the set of metrics needed by QOOD. From the measurements it generates measurement reports that can be imported in a spreadsheet for further processing. Alternatively, HTML reports for design reviews can be generated for classes, interfaces, packages, and the system. These reports contain an overview of the element, its measurements and additional checklists for aspects that cannot be measured on ODEM, especially semantic issues like cohesion or consistency.

2. XMI (XML Metadata Interchange; see [OMG00b]) is the OMG standard for expressing UML models in XML. It is supposed to become the standard model interchange format for UML tools.

References

- [Ab01] Abreu, F.: Using OCL to formalize object-oriented metrics definitions. In (Abreu, F. et al., eds.): Proceedings of the 5th Int. ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (Budapest, June 2001), 99-134.
- [Bo76] Boehm, B. (1976): Software Engineering. IEEE Transactions on Computers 25(12), 1226-1241.
- [BB01] Boehm, B. ; Basili, V.: Software Defect Reduction Top 10 List. IEEE Computer 34(1), 2001; S. 135-137.
- [CG90] Card, D.; Glass, R.: Measuring Software Design Quality. Prentice-Hall, Englewood Cliffs, NJ, 1990.
- [CK94] Chidamber, S.; Kemerer, C.: A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering 20(6), 1994; S. 476-493.
- [Ma95] Martin, R.: Designing Object-Oriented C++ Applications Using the Booch Method. Prentice Hall, Englewood Cliffs, NJ, 1995.
- [OMG00a] Object Management Group: OMG Unified Modeling Language Specification, Version 1.3, March 2000.
- [OMG00b] Object Management Group: OMG XML Metadata Interchange (XMI) Specification, Version 1.1, November 2000.
- [Re00] Reißing, R.: Ein Qualitätsmodell für den objektorientierten Entwurf. In (Gesellschaft für Informatik, Hrsg.): Informatiktage 2000, 27. und 28. Oktober 2000 im Neuen Kloster Bad Schussenried. Konradin Verlag, 2000; S. 154-157.
- [Re01a] Reißing, R.: Towards a Model for Object-Oriented Design Measurement. In (Abreu, F. et al., eds.): Proceedings of the 5th Int. ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (Budapest, June 2001); S. 71-84.
- [Re01b] Reißing, R.: Assessing the Quality of Object-Oriented Designs. Proceedings of the Doctoral Symposium at OOPSLA'01 (Tampa, FL, October 2001), OOPSLA'01 Companion, 2001.
- [Ro90] Rombach, H.: Design Measurement: Some Lessons Learned. IEEE Software 7(2), 1990; S. 17-25.
- [WK99] Warmer, J.; Kleppe, A.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley, Boston, 1999.