Towards a Model for Object-Oriented Design Measurement

Ralf Reißing
Institute of Computer Science, University of Stuttgart
Breitwiesenstr. 20-22
70565 Stuttgart, Germany
reissing@informatik.uni-stuttgart.de

Abstract

Object-oriented design plays 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, in order to automate design evaluation a precise definition of metrics is needed. 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. This model can serve as a foundation for the formal definition of object-oriented design metrics. ODEM 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 of the use of ODEM for defining object-oriented metrics are given. Two case studies on existing metrics suites for object-oriented design show the benefits of applying ODEM to established object-oriented design metrics.

1 Introduction

Design is an important cost driver in software development, for it does not only cause the cost of its own creation, but it also heavily influences the cost of the following phases, i.e. implementation and maintenance. The design phase only takes 5-10% of the total effort (over the whole software life-cycle), but a large part (up to 80%) of the total effort goes into correcting bad design decisions [1]. If bad design is not fixed in the design phase, the cost for fixing it after delivery of the software is between 5 and 100 times higher [3].

But even if the initial design is good enough for the moment, there may be difficulties when trying to make changes and extensions in the maintenance phase. The characteristics of the design, e.g. flexibility, heavily influence the ease of maintenance. As at least 50% of the total life-cycle cost goes into maintenance [2], it is very cost effective to have a good design early and to maintain high design quality throughout the life-cycle.

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 [4][15]. 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. Early design measurement means, of course, that it should not depend on detailed design information or even on code.

Design measurement in itself, i.e. without a purpose, is useless because the actual measurements would have no meaning. 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.

Design evaluation can be done by experts using checklists which are based on the quality model and its metrics. However, typical designs are so big that quality assurance becomes a time-consuming activity. Therefore it is more efficient to use an automated tool for design evaluation. Even though such a tool can never replace a human expert, it can help him (or her) to identify components of a design 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. Most metrics proposed so far do not have such a formal definition (see section 2.1), so a formal model of design is needed as a reference model for precise metric definition (see section 2.2). Such a formal model called ODEM is presented in depth in section 3. Examples for its use for metric definition are given in section 4. In two case studies in section 5, ODEM is applied to established metrics suites for object-oriented design by Martin and by Chidamber and Kemerer. Finally, as an outlook extensions to the model, and its use in design evaluation are discussed.

2 Object-Oriented Design Metrics

2.1 State of the Art

Up to now, a large number of metrics for object-oriented systems has been proposed. Whole collections of metrics were published, e.g. by Lorenz and Kidd [11] and by Henderson-Sellers [9]. Most well-known is the metrics suite proposed by Chidamber and Kemerer [5][6].

A major problem of many of these metrics is their lack of precise definition. For example, Li and Henry [10] define their metric NOM (number of methods) as “the number of local methods” of a class. Unfortunately they do not define the term “local method”. From the context it can be guessed that inherited methods are not counted. But what about class methods? Redefined methods? Is the method visibility (public etc.) considered? Many questions are left unanswered, nevertheless the authors validate their metric as a predictor for maintenance effort.

As long as there is no precise definition of a metric, measurement of the metric is not repeatable. Any empirical validation of the metric’s predictive value for quality attributes is useless because the actual measurement depends on the person who measures. Somebody using the metric might interpret the ambiguous definition differently and therefore measure something other than the creator of the metric intended – so the original validation does not apply anymore. Also, the experiments for metric validation cannot be reproduced if the metric is not defined precisely. Hence, a precise, formal definition is a prerequisite for serious measurement.

Another problem is the fact that many metrics (e.g. method complexity or number of private methods) are not suited for early design measurement because their definition depends on code. Without code (or a very detailed design) these metrics are useless. Because design quality is already important in the design phase, design measurement should be possible without code.

2.2 Suggested Solution

The problems outlined above can be solved. First, the requirements to the solution are given. Based on the requirements the proposed solution, a formal model of object-oriented design, is outlined.

2.2.1 Requirements

The observations in section 2.1 lead to a statement of requirements for metrics of object-oriented design:

1. The definitions of metrics shall be precise and unambiguous (this, however, is a goal that is not achieved easily [8]).
2. The metrics shall be based on design artefacts that are typically constructed in the design phase.
3. The metrics shall be suited for automatic measurement as far as possible.
4. Measurement shall be possible even if the design is incomplete or not detailed.

2.2.2 Solution

The above requirements can be fulfilled by doing the following:

1. A formal model of object-oriented design is introduced.
The formal model defines precisely which attributes the design has. The metrics can then be expressed precisely and unambiguously in terms of the formal model.
2. The formal model captures the elements of the Unified Modeling Language (UML) class and package diagrams.
Design artefacts are expressed using a notation – the UML is the most popular and therefore most widespread notation. When documenting a design with UML, the only diagram types consistently used in practice are the class and package diagrams.
3. The formal model for metric definition is based on the UML meta-model.
The UML meta-model provides a formal model of object-oriented design in UML. The XML Metadata Interchange (XMI), a standard notation for UML model descriptions, is based on it. Most UML tools can produce XMI files, therefore, XMI is a suitable input format for automatic measurement.
4. The focus of the formal model is on high-level design.
Thus, design measurement does not depend on detailed design information and can already be done early in the design phase.

3 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. The word “Odem” is an ancient German word for breath. This is a very suitable name because having a formal model is as vital for the precise definition of metrics as breathing is for living.

ODEM is based on the UML meta-model, namely the part of the meta-model needed to capture class and package 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 make metric definitions easier to read and understand. The resulting structure is depicted in figure 1.

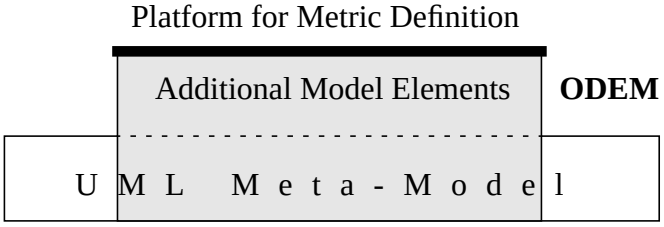

Figure 1: Conceptual structure of ODEM

3.1 The UML Meta-Model

For the interested reader this section gives an overview of the UML meta-model. For the understanding of the rest of the paper, a full understanding of the UML meta-model is not necessary because the abstraction layer in ODEM (described in section 3.3 and the following) tries to hide it as far as possible.

The UML meta-model is defined in the UML specification by the Object Management Group (OMG) [13] and provides a formal definition for the UML. Interestingly, the UML meta-model is again defined using the UML (recursive definition!). Because the UML is an object-oriented design notation, the elements of the UML meta-model are of course modeled as classes. The names of the classes reflect the corresponding UML terms.

Figure 2 shows the UML class diagram for the model elements (class, package, etc.) in ODEM. Figure 3 depicts the relationships (association, dependency, etc.) between the model elements.

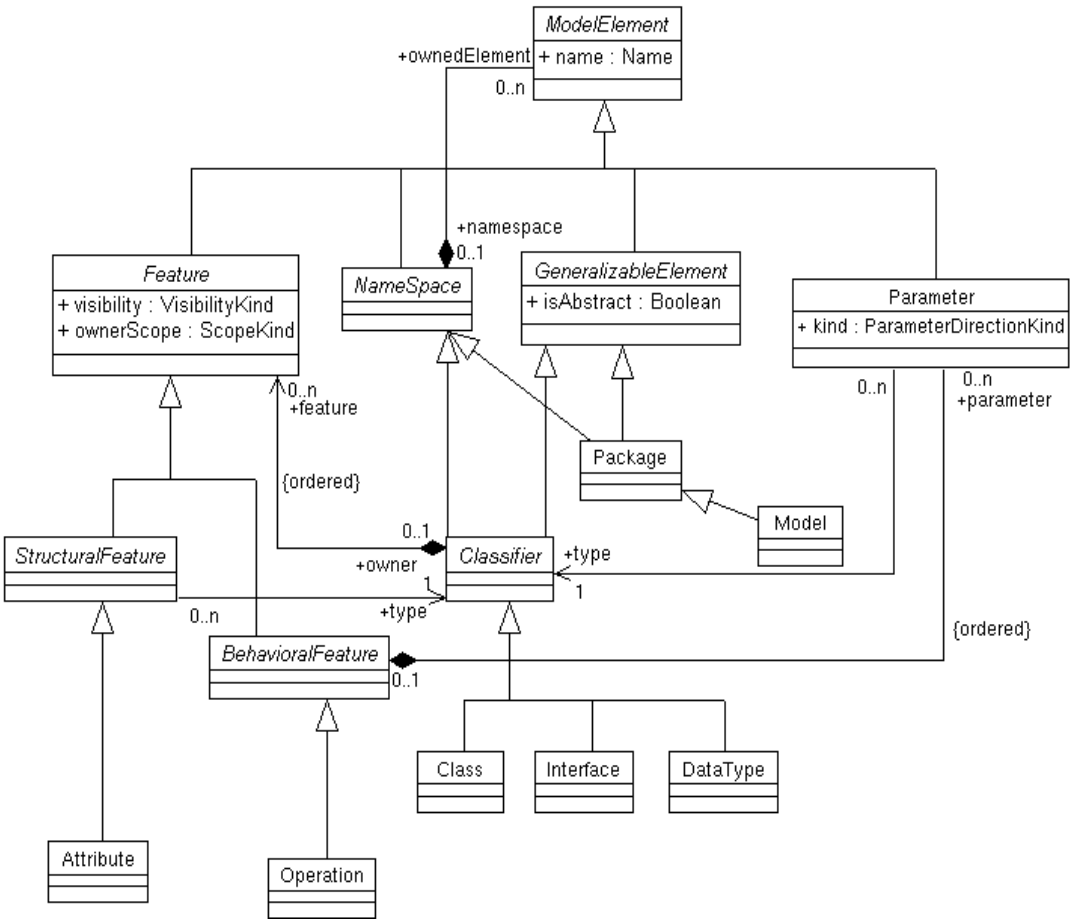

Figure 2: ODEM model elements

The common superclass of all model elements is the class ModelElement (see figure 2). NameSpace, a subclass of ModelElement, contains an arbitrary number of ModelElements and is used for hierarchical structuring. Package and Model are subclasses of NameSpace. A model is a special package that contains everything belonging to a UML model.

The class Classifier is a subclass of GeneralizableElement, so it can inherit features from other classifiers. Classifiers can have Features (modeled by a composition), e.g. Attributes and Operations. Class, Interface, and DataType are subclasses of Classifier.

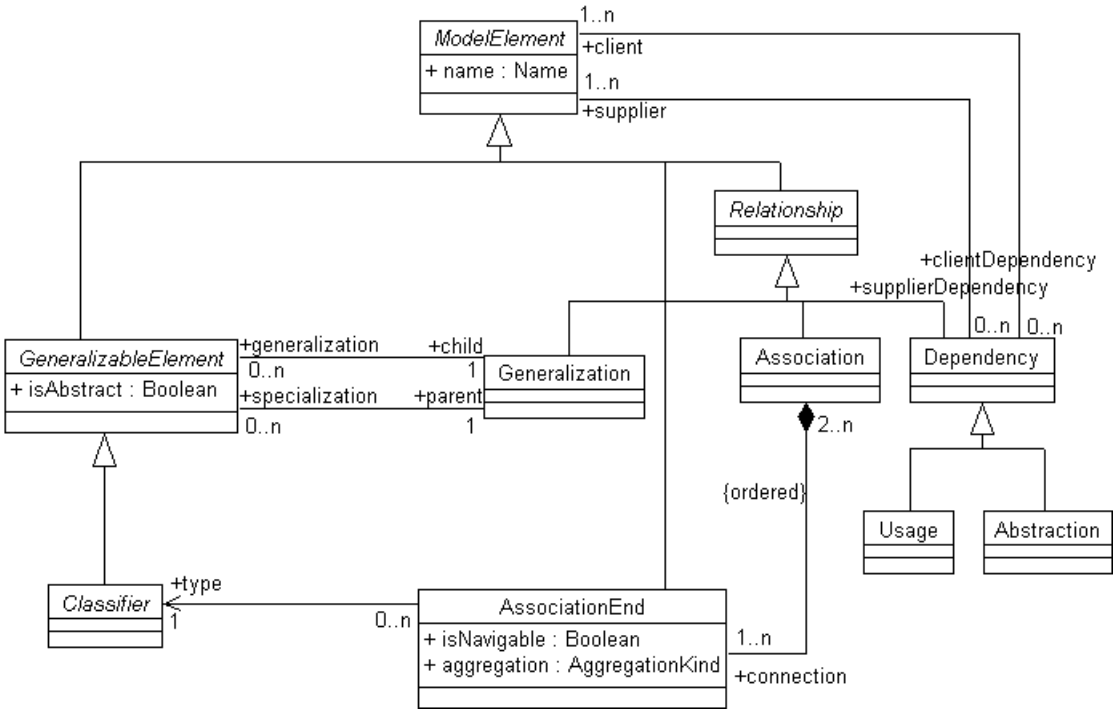

Figure 3: ODEM relationships

Relationships between model elements like classes and interfaces are also modeled as classes (see figure 3). These classes are subclasses of Relationship, a subclass of ModelElement. Associations are modeled using the extra class AssociationEnd because an Association can connect more than two Classifiers, whereas the other relationships always connect exactly two Classifiers.

The UML specification also defines the Object Constraint Language (OCL) as a part of the UML. The OCL is a formal language for expressing constraints on UML models. It was tried to use the OCL as a specification language for design metrics, but it does 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.

3.2 Scope of ODEM

The focus of ODEM is on high-level design. Therefore, methods, i.e. implementations of operations, are not included – from the viewpoint of high-level design they are an implementation detail because they are not part of the interface of a class.

Currently ODEM is restricted to the structural aspects of the UML, especially class and package diagrams, because these diagrams are most likely to be present and complete in typical design descriptions. The model elements have been reduced to a minimum: For example, template classes and inner classes (i.e. classes nested into classes) are not featured. These restrictions may seem somehow arbitrary, but a line must be drawn somewhere to keep the model size manageable.

The additional model elements are introduced in the following subsections on the system, packages, interfaces, classes, attributes, operations, and parameters. Each subsection shows the attributes and the relationships (if any) of the respective model elements. Table 1 shows an overview of the formal identifiers used.

Every model element has an attribute called name (inherited from class ModelElement). This name is used to identify model elements. The name attribute is not relevant for the definition of

metrics, but it is essential for the output and presentation of metric values. For the sake of brevity, the name attribute is not listed in the following sections.

Table 1: Overview of formal identifiers in ODEM

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
associates*	Extended associates relation (includes inherited associations)
contains	Relation for container relationship between a package and a model element
contains*	Extended contains relation (transitive closure)
depends_on	Aggregate relation for relationships between classes/interfaces that cause a dependency
depends_on*	Extended depends_on relation (includes inherited dependencies)
extends	Relation for inheritance of features
extends*	Extended extends relation (transitive closure)
has	Relation for the containment of features in classes/interfaces
has*	Extended has relation (includes inherited attributes/operations)
realizes	Relation for realization of an interface by a class
realizes*	Extended realizes relation (includes inherited realizations)
uses	Relation for use of a class/interface by a class/interface
uses*	Extended uses relation (includes inherited uses)

3.3 System

The designed system S consists of packages. S itself is a special package, therefore the characteristics of packages defined in section 3.4 also apply to S. S is the only instance of Model (see figure 2), a subclass of NameSpace and Package.

3.4 Package

Packages group classes, interfaces and (nested) packages. A package is an instance of the class Package (see figure 2).

3.4.1 Relationships

contains: $P \times (P \cup C \cup I)$

The composition of ModelElement in NameSpace (see figure 2) can be expressed as a relation: *contains*(p,q) iff there is a composition of p (role namespace) and q (role ownedElement). Each package, class, and interface must be contained in exactly one package –with the exception of S that is not contained in any package.

3.5 Interface

An interface is a set of operations. It can be considered to be a special case of an abstract class without attributes. Nevertheless it is useful to differentiate between interfaces and classes because in some object-oriented programming languages (e.g. Java) the difference is significant. An interface is an instance of the class Interface (see figure 2).

3.5.1 Attributes

isAbstract: Boolean.
Always true because interfaces are always abstract.

3.5.2 Relationships

extends: $I \times I$
The class Generalization (see figure 3) represents inheritance (or extension) relationships between GeneralizableElements (Interface is a subclass of GeneralizableElement, see figure 2): *extends*(i,j) iff there is an instance g of Generalization where g.parent=i and g.child=j. An interface can extend an arbitrary number of other interfaces.

has: $I \times O$
The composition of Operation (a Feature subclass, see figure 2) in Interface (a Classifier subclass) can be expressed as a relation: *has*(i,o) iff there is a composition of i (role owner) and o (role feature). Inherited operations are not considered in the relation. An interface can have an arbitrary number of operations.

uses: $I \times (C \cup I)$
The class Usage (a Dependency subclass, see figure 3) represents usage relationships between ModelElements: *uses*(i,j) iff there is an instance u of Usage where u.client=i and u.supplier=j. An interface can use an arbitrary number of classes and interfaces (by using them as a parameter type of an operation).

3.6 Class

A class is a set of attributes and operations. A class is an instance of the class Class (see figure 2).

3.6.1 Attributes

isAbstract: Boolean

3.6.2 Relationships

extends: $C \times C$
The class Generalization (see figure 3) represents inheritance (or extension) relationships between GeneralizableElements (GeneralizableElement is a superclass of Class, see figure 2): *extends*(c,d) iff there is an instance g of Generalization where g.parent=c and g.child=d. A class can extend an arbitrary number of other classes (multiple inheritance).

realizes: $C \times I$
The class Abstraction (a Dependency subclass, see figure 3) represents realization relationships between classes and interfaces: *realizes*(c,i) iff there is an instance a of Abstraction with the stereotype «realize» where a.client=c and a.supplier=i. A class can realize an arbitrary number of interfaces.

has: $C \times A$
The composition of Attribute (a Feature subclass, see figure 2) in Class (a Classifier subclass) can be expressed as a relation: *has*(c,a) iff there is a composition of c (role owner) and a (role feature).

Inherited attributes are not considered in the relation. A class can have an arbitrary number of attributes.

has: $C \times O$

The composition of Operation in Class can be expressed as a relation: *has*(c,o) iff there is a composition of c (role owner) and o (role feature). Inherited operations are not considered in the relation. A class can have an arbitrary number of operations.

associates: $C \times (C \cup I)$

The class Association (see figure 3) together with the class AssociationEnd represents associations between Classifiers. An Association has at least two AssociationEnds that store association attributes like navigability (attribute isNavigable) and the kind of association (attribute aggregation; values: none, aggregate, composite). *associates*(c,d) iff there is an instance a of Association that contains two instances e1 and e2 of AssociationEnd where e1.type=c and e2.type=d and e2.isNavigable=true (the latter requires the association to be navigable, which may prove to be too restrictive). e1.aggregation determines the kind of association. A class can associate an arbitrary number of classes and interfaces.

uses: $C \times (C \cup I)$

The class Usage represents usage relationships between ModelElements: *uses*(c,d) iff there is an instance u of Usage where u.client=c and u.supplier=d. A class can use an arbitrary number of classes and interfaces (e.g. by using them types of attribute or parameter types of an operations, or by calling an operation).

3.7 Attribute

An attribute is an instance of the class Attribute (a Feature subclass, see figure 2).

3.7.1 Attributes

type: Classifier.

The type of the attribute is determined by the directed association of StructuralFeature with Classifier (role type).

visibility: VisibilityKind.

The visibility of the attribute: public, protected, or private.

ownerScope: ScopeKind. The scope of the attribute: classifier (i.e. class attribute) or instance (i.e. instance attribute)

3.8 Operation

An operation is an instance of the class Operation (a Feature subclass, see figure 2).

3.8.1 Attributes

visibility: VisibilityKind.

The visibility of the operation: public, protected, or private.

ownerScope: ScopeKind.

The scope of the operation: classifier (i.e. class attribute) or instance (i.e. instance attribute).

3.8.2 Relationships

has: $O \times M$

The composition of Parameter (see figure 2) in BehavioralFeature (the Operation superclass) can be expressed as a relation: *has*(o,p) iff there is a composition of o (role type) and p (role parameter). An operation can have an arbitrary number of parameters. The return type of an operation is expressed as a special parameter (see section 3.9).

3.9 Parameter

A parameter is an instance of the class Parameter (see figure 2).

3.9.1 Attributes

kind: ParameterDirectionKind.

The kind of the parameter: in, out, inout, or return. The first three values signify the direction of the parameter from the operation's perspective. The value return signifies a pseudo-parameter used to store the return type of an operation. An operation does not need to have a return type.

type: Classifier.

The type of the parameter is determined by the directed association of Parameter with Classifier (role type).

3.10 Extensions to the Model

3.10.1Additional Relations

So far, the relations like *has*, *associates*, and *uses* 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:

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

Now the other extended relations (*associates**, *realizes**, *has**, and *uses**) can be defined based on *extended**. For example, the relation *uses** is defined as follows:

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

Now, a general dependency relation called *depends_on* is introduced to capture all kinds of dependency between model elements: extension, realization, association, and use. Note that it is assumed here that all kinds of use (like use by operation call, attribute type, or parameter type) were properly modeled in the design description by a Usage instance (such instances could be added automatically for the last two example cases if necessary).

depends_on(x,y) iff *extends*(x,y) $\vee$ *realizes*(x,y) $\vee$ *associates*(x,y) $\vee$ *uses*(x,y).

The extended relation *depends_on** is defined similar to the other *-relations.

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

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

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

3.10.2Weighted 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 default weight of each 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 as an aggregate relation has a weight that is the sum of the weights of the relations included.

For the *-relations, the weight of inherited relations has to be taken into account, too, which is anything but trivial except for *contains**, which always has a weight of 1. For *extends** the weight is the number of inheritance paths between the first and the second model element. 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.

4 Metrics for Object-Oriented Design

Using ODEM, design metrics can be defined. Here only a few examples are given to demonstrate how the metric definition with ODEM works. There are three categories: class, package, and system metrics. Each example metric in this section is introduced along with a possible use for it to motivate the need for the metric.

4.1 Class Metrics

There is a heuristics that discourages having public attributes in a class (for better encapsulation). The metric NAP (number of attributes in the public interface of a class) captures the necessary information:

NAP(c) = |{a∈ A: *has**(c,a) ∧ a.*visibility* = public}|

A similar heuristic states that the number of attributes visible to subclasses should be minimized. The metric NAI (number of attributes in the inheritance interface of a class) is defined:

NAI(c) = |{a∈ A: *has**(c,a) ∧ (a.*visibility*∈ {public, protected})}|

A fundamental principle of design is to reduce coupling. One kind of coupling is coupling by association. It is possible to distinguish between local and inherited associations.

NIA (number of inherited associations of a class)
NIA(c) = |{c'∈ C: ∃c"∈ C: *extends**(c,c") ∧ *associates*(c",c')}|

NLA (number of local associations of a class)
NLA(c) = |{c'∈ C: *associates*(c,c')}|

NAA (number of all associations of a class)
NAA(c) = NIA(c) + NLA(c)

Note that the last three metrics do not take into account that there may be more than one association between classes. If this distinction is necessary, instead of counting the relations their weights have to be summed up.

4.2 Package Metrics

A heuristic states that the nesting level of containment hierarchies should not be too deep, say 5 to 7 maximum. The metric DNH (depth in the nesting hierarchy, i.e. number of containing packages) can measure this. A recursive definition is the most simple one:

DNH(S) = 0 (so top level packages have a DNH of 1)
DNH(p) = DNH(p': *contains*(p',p)) + 1

Also, the number of classes and nested packages should not be too high.

NCP (number of all classes in a package)
NCP(p) = |{c∈ C: *contains*(p,c)}|

NPP (number of nested packages in a package)
NPP(p) = |{p'∈ P: *contains*(p,p')}|

4.3 System Metrics

System metrics are mostly aggregate metrics, e.g. the total number of classes in the system, the mean number of methods per class or the maximum depth of the inheritance hierarchies.

Nevertheless there are some metrics original to the system. For example, the metric NIH (number of inheritance hierarchies, i.e. number of root classes) should not be too high, else this hints to poor use of inheritance.

NIH(S) = |{c∈ C: ¬∃c'∈ C: *extends*(c,c')}|

5 Case Studies

In order to test the applicability of ODEM to metrics already defined in the literature, both Martin's package metrics [12] and the well-known Chidamber and Kemerer metrics suite [6] are formalized using ODEM.

5.1 Martin's Package Metrics

Martin identifies criteria for the proper distribution of classes into packages. These criteria are essentially based on the notion of dependency. The goal is to reduce dependency, especially dependency on concrete classes. Unfortunately, Martin does not define what a dependency exactly is. He only says that dependencies are caused by class relationships like inheritance, aggregation, and use. As an educated guess the depends_on-relation, which includes the examples given by Martin, is used for the formal definitions.

Martin does not consider nested packages, even though dependencies of classes in packages nested inside a package to classes within that package can be considered to have a special status, as they are more "local" than dependencies from classes in outside packages.

5.1.1 The Metrics

1. Relational cohesion (H). This metric wants to capture the cohesion of classes inside a package. As classes inside a package should be strongly related, the cohesion should be high. H is defined as (R+1)/N where R is the number of relationships between classes in a package and N is the number of classes in the package.

$H(p) = (\Sigma_{c \in C:\ contains(p,c)}\ NID(c) + 1) / NCP(p)$

where NID (number of internal dependencies of a class in a package) is defined as:

$NID(p,c) = \text{if } contains(p,c) \text{ then } \Sigma_{c' \in C \cup I:\ contains(p,c')}\ depends\_on(c,c').weight \text{ else } 0 \text{ endif}$

Note that this definition counts a mutual dependency twice, once for each direction.

2. Afferent Coupling (Ca). The number of classes from other packages that depend on the classes within the package. Afferent coupling should be low.

$Ca(p) = |\{c \in C \cup I: \neg contains(p,c) \wedge (\exists c' \in C \cup I: contains(p,c') \wedge depends\_on(c,c'))\}|$

3. Efferent Coupling (Ce). The number of classes from other packages that the classes within the package depend upon. Efferent coupling should be low.

$Ce(p) = |\{c \in C \cup I: \neg contains(p,c) \wedge (\exists c' \in C \cup I: contains(p,c') \wedge depends\_on(c',c))\}|$

4. Abstractness (A). The abstractness of a package is defined as the ratio of abstract classes to the total number of classes. Martin does not mention interfaces, so they are counted as abstract classes here. The special case of a package containing no classes (not mentioned by Martin) is considered to have an abstractness of 1. Such a package is still meaningful if used for structuring packages.

$A(p) = \text{if } NCP(p) > 0 \text{ then } (NACP(p) + NIP(p)) / NCP(p) \text{ else } 1 \text{ endif}$

where NACP (number of abstract classes in a package) is defined as:

$NACP(p) = |\{c \in C: contains(p,c) \wedge c.isAbstract\}|$

and NIP (number of interfaces in a package) is:

$NIP(p) = |\{i \in I: contains(p,i)\}|$

5. Instability (I). The instability of a package is defined as the ratio of efferent coupling to total coupling. The special case of a package coupled to no external classes (not mentioned by Martin) is considered to have an instability of 0.

$I(p) = \text{if } Ca+Ce > 0 \text{ then } Ce / (Ca+Ce) \text{ else } 0 \text{ endif}$

6. Distance from the Main Sequence (D). The main sequence is part of a theory of Martin that states that the abstractness A and the instability I of a package should be about the same. That is, abstractions have to be very stable, concrete implementations may change more. The more distant a package is from the main sequence, expressed as A+I=1, the worse. There is also a normalized version D' that has a [0,1] range.

$D(p) = |A(p) + I(p) - 1| / \sqrt{2}$

$D'(p) = |A(p) + I(p) - 1| = \sqrt{2}\ D(p)$

5.1.2 Results

Martin's metrics focus on high-level, architectural design issues, so they can be formalized easily. There are some vague points in the original definition, but these could be overcome in the formalization by educated guesses.

5.2 Chidamber and Kemerer's Metrics Suite

Chidamber and Kemerer were among the first to publish a set of metrics for object-oriented design. Here the second version of their six metrics, as published in [6], is used for formalization.

5.2.1 The Metrics

1. Weighted Methods per Class (WMC). WMC is defined as the sum of the complexities of the methods of a class. The definition of complexity is left open deliberately as an implementation detail. Generally WMC is used with a standard complexity of one, so it is essentially used as a method count, even by the authors themselves (see [7]). A lower WMC is better.

Unfortunately, the authors do not state which methods should be counted, e.g. public methods only or local methods only. The main interpretation in the literature seems to be counting all methods that were locally defined (including redefinitions). However, ODEM considers operations, not methods, so redefinitions cannot be counted.

$WMC(c) = |\{o \in O: has(c,o)\}|$

Note that this definition makes no difference between class and instance methods.

2. Depth of Inheritance Tree (DIT). The authors defined the DIT of a class as the maximum length of all inheritance paths from the class to the root class of its inheritance hierarchy. A lower DIT is better, but a low mean DIT may hint to little reuse by inheritance. Here a recursive definition is used:

$DIT(c) = \text{if } NSC(c) > 0 \text{ then } \max_{c' \in C:\ extends(c,c')} \{DIT(c') + 1\} \text{ else } 0 \text{ endif}$

where NSC (number of (direct) superclasses) is defined as:

$NSC(c) = |\{c' \in C: extends(c,c')\}|$

3. Number of Children (NOC). The number of immediate subclasses of a class in the class hierarchy. A high value may indicate both better reuse, misuse of subclassing, and higher testing effort, so there is no clear statement which values indicate a better design.

$NOC(c) = |\{c' \in C: extends(c',c)\}|$

4. Coupling between Object Classes (CBO). The number of other classes a class is coupled to. According to the authors a class A is coupled to a class B if a method of A uses a method or an instance variable of B. Low coupling is better.

This metric requires detailed information about the methods, which means that a detailed design description or the class code is necessary for measurement. ODEM as a high-level design model does not provide this information.

5. Response for a Class (RFC). The size of the response set of a class, i.e. the number of methods of a class plus the number of methods of other classes used by the methods of the class (each method is counted only once). A small response set is better.

Again, this metric requires detailed information about the methods, so the problem is the same as with CBO. Additionally, the authors do not say which methods in the class are to be considered for RFC, e.g. if inherited methods or class methods count.

6. Lack of Cohesion in Methods (LCOM). This metric tries to capture the cohesion of methods by considering similarities. The similarity of two methods is defined by the number of instance variables both methods use. LCOM is the defined as the number of methods pairs with a similarity of 0 (no common instance variable) less the number of pairs with a similarity greater 0. If the difference is less than 0, LCOM is 0. The lower LCOM, the better.

This metrics requires information about the use of instance variable by methods, which is also detailed design information.

5.2.2 Results

DIT and NOC can be formalized easily because these metrics apply to the inheritance hierarchy which is part of the high-level design. The metrics CBO, RFC, and LCOM require detailed design knowledge for measurement. WMC also needs detailed design knowledge (for the method complexity) and is defined too vaguely to allow a proper formalization. As these metrics are obviously intended as low level design metrics, trying to formalize them with ODEM is useless.

The model elements needed to formalize CBO, RFC, and LCOM could be added to ODEM. The necessary addition of a uses relation between operations or between operations and attributes is theoretically possible: Attributes and Operations are ModelElements, so the formal model could contain usage dependencies between them. However, it does not seem realistic that somebody would add dependencies between operations to a class diagram. This is because the UML, the foundation of ODEM, is rarely used as a detailed design notation. So the information would simply not be available, or only if it were extracted by a tool from code.

Detailed information about method behavior is partially available in the UML, e.g. in activity and state diagrams, or in the scenarios of class cooperation in sequence and collaboration diagrams. Nevertheless, even this information is not sufficient to calculate WMC, CBO, RFC, and LCOM.

6 Outlook

6.1 Future Extensions

The case studies show that ODEM meets it purpose as a foundation for structural high-level design measurement. Its limits of applicability to detailed design measurement are due to the deliberate restrictions of the formal model.

Future extensions of the formal model can incorporate the missing elements from the UML meta-model for the class diagram features, e.g. template classes. To include dynamic information, the full UML meta-model might be used in order to also access the information from action, state, and interaction diagrams. An alternative is to use code (if available) for the extraction of detailed design information.

6.2 Automation of Measurement

ODEM is used as the foundation for the definition of the metrics in the quality model QOOD (Quality of Object-Oriented Design, see [14]), which is currently under development. For higher comfort of design quality evaluation with QOOD, a tool is needed.

Because ODEM is based on the UML meta-model, implementing tools that calculate metrics defined with ODEM should be easy. The implementation can be based on a tool that parses XMI model descriptions generated by a UML tool. The functionality can also be directly integrated in a UML tool if it supports scripting (e.g. Rational Rose) or if it is open source (e.g. ArgoUML). If only code is available, design information can be reverse-engineered from the code first by a UML tool.

References

[1] Bell, G.; Morrey, I.; Pugh, J.: Software Engineering: A Programming Approach. Prentice Hall, New York, 1987.

[2] Boehm, B. (1976): Software Engineering. IEEE Transactions on Computers, 25(12), 1226-1241.

[3] Boehm, B. ; Basili, V.: Software Defect Reduction Top 10 List. IEEE Computer, 34(1), 2001, 135-137.

[4] Card, D.; Glass, R.: Measuring Software Design Quality. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[5] Chidamber, S.; Kemerer, C.: Towards a Metrics Suite for Object Oriented Design. Proceedings of OOPSLA'91, ACM SIGPLAN Notices, 26 (11), 1991, 197-211.

[6] Chidamber, S.; Kemerer, C.: A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, 20(6), 1994, 476-493.

[7] Chidamber, S.; Darcy, D.; Kemerer, C.: Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis. IEEE Transactions on Software Engineering, 24(8), 1998, 629-639.

[8] Churcher, N.; Shepperd, M.: Towards a Conceptual Framework for Object-Oriented Software Metrics. ACM SIGSOFT Software Engineering Notes, 20(2), 1995, 69-76.

[9] Henderson-Sellers, B.: Object-Oriented Metrics. Prentice Hall, Englewood Cliffs, NJ, 1996.

[10]Li, W.; Henry, S.: Object-Oriented Metrics that Predict Maintainability. Journal of Systems and Software, 23(2), 1993, 111-122.

[11]Lorenz, M.; Kidd, J.: Object-Oriented Software Metrics: A Practical Guide. Prentice Hall, Engelwood Cliffs, NJ, 1994.

[12]Martin, R.: Designing Object-Oriented C++ Applications Using the Booch Method. Prentice Hall, Englewood Cliffs, NJ, 1995.

[13]Object Management Group: OMG Unified Modeling Language Specification, Version 1.3, March 2000.

[14]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, 154-157.

[15]Rombach, H.: Design Measurement: Some Lessons Learned. IEEE Software 7(2), 1990, 17-25.