Chidamber & Kemerer's Metrics Suite: A Measurement Theory Perspective

Martin Hitz, Behzad Montazeri

Institut für Angewandte Informatik und Informationssysteme
University of Vienna
Rathausstraße 19/4
A-1010 Vienna
Austria / Europe

email: {hitz, montazeri}@ifs.univie.ac.at

Abstract

The metrics suite for object-oriented design put forward by Chidamber and Kemerer [8] is partly evaluated by applying principles of measurement theory. Using the object coupling measure (CBO) as an example, it is shown that failing to establish a sound empirical relation system can lead to deficiencies of software metrics. Similarly, for the object-oriented cohesion measure (LCOM) it is pointed out that the issue of empirical testing the representation condition must not be ignored, even if other validation principles are carefully obeyed. As a by-product, an alternative formulation for LCOM is proposed.

Index Terms

Software Measurement, Coupling Metrics, Cohesion Metrics, Object-Orientation,Validation

1. Introduction

Chidamber and Kemerer (“C&K” in the remainder of this article) proposed a draft suite of software metrics tailored to object-oriented software design in 1991, featuring several internal[1] product attributes [7]. In a more recent article, the authors present a more thorough treatise on their candidate metrics, complemented by the analysis of two empirical studies and a short guideline of how to apply their metrics to support the object-oriented design process [8]. After a short review of the most important aspects of measure theory pertinent to software metrics as put forward by, e.g., Fenton [10][11], the authors link their metrics to an ontological foundation [4][5] and validate them according to a subset of Weyuker's axioms [23]. Specifically, the metrics suggested are Weighted Methods Per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling between Object Classes (CBO), Response For a Class (RFC), and Lack of Cohesion in Methods (LCOM).

---

[1] *Internal* product attributes can be measured purely in terms of the product itself as opposed to *external* attributes which can only be measured with respect to how the product relates to its environment [10].

A closer look on the proposed metrics reveals some shortcomings which can be detected (and avoided) by rigorously applying simple measure theoretic principles[2]. In general, as has been pointed out be several authors, measurement (in the assessment sense) consists of the following activities [1][10][11]:

- Identifying the attribute of interest,
- establishing an empirical relation system,
- finding a measure mapping the empirical relation system into a formal (numerical) one,
- validating the measure, and
- determining the scale type of the measure.

We show how more careful consideration of measurement theory highlights both the strengths and weaknesses of some of the proposed metrics.

2. The importance of identifying attributes and empirical relation systems

Before any measurement activity, we must identify the attribute to be measured. Such an attribute must bear a certain *significance* for a person involved in the development process, such as a designer, programmer, manager, user etc. The attribute might not necessarily be interesting *per se*, but might serve as an independent variable for indirect measurement of another (interesting!) attribute or in a given prediction model (a case we do not consider further here), however, one should avoid collecting data about meaningless aspects of the software document under investigation (just because they happen to be easily collectible).

In the next step, a "sufficient" empirical relation system must be established, i.e., an empirical relation system which captures all generally accepted intuitive ideas about the attribute under consideration. In what follows, we demonstrate the consequences of not strictly adhering to this rule by means of a concrete coupling metric taken from [8]. Coupling is one of the most famous internal product attributes, studied since the advent of structured programming [18][22][26] and more recently, in the context of object-orientation [3][13][14][16][24]. Under the object-oriented paradigm, the notion of coupling differs somewhat from the classical one, but the main ideas remain the same.

In [8], *Coupling Between Object Classes* (CBO) is defined as "a count of the number of other classes to which it is coupled". This definition implies that all single couples as defined in [8] ("two classes are coupled when methods declared in one class use methods or instance variables of the other class") are considered equal. However, a more complete empirical relation system would demand that, depending on several circumstances, a single couple should contribute more or less to an overall coupling measure. Specifically, one should consider at least the following empirical relations (where applicable) [12]:

- Access to instance variables constitutes stronger coupling than pure message passing.
- Access to instance variables of foreign classes constitutes stronger coupling than access to instance variables of superclasess (the same holds, *mutatis mutandis*, for message passing).
- Passing a message with a wide parameter interface yields stronger coupling than passing one with a slim interface.
- Violating the Law of Demeter [16][17] yields stronger coupling than restricted (Demeter conforming) message passing[3].
- Couples to the following types of objects should yield increasing coupling values: local objects, method parameters, subobjects of self (= instance variables of class type), subobjects of a super class

---

[2] Recently, another comment on the work of C&K has been published by Churcher and Shepperd who argue that it is most important to specify the mapping from a language-independent set of metrics to specific programming languages [9]. Although we agree with this observation, here we are focusing on other issues.

[3] In [16], the Law of Demeter in its so-called "object version" is defined for C++ as follows:
*For all classes C, and for all member functions M attached to C, all objects to which M sends a message must be: 1. M's argument objects, including *this or 2. a data member object of class C.*

(= inherited instance variables of class type), global objects.

Of course, it is debateable, which of the above rules are to be taken into consideration. However, as C&K explicitly refer to the Law of Demeter in the context of coupling, at least the fourth relationship above should have been captured by their metric. For instance, both designs presented in Fig. 1 result in CBO(C) = 2, because method M of class C uses methods of classes A and B in both cases. However, the Law of demeter is violated in the left example in Fig. 1 due to the “indirect” method dispatch b->f()**->g()** which is avoided in the right example by means of a specific access method in B.

```
class A { ... void g(); ...};
class B { ... A* f(); ... };
class C {
        A* a; B* b;
        void M () { a->g(); b->f()->g(); }
};
```

```
class A { ... void g(); ...};
class B { ... A* f(); void apply_g(); ... };
class C {
        A* a; B* b;
        void M () { a->g(); b->apply_g(); }
};
```

Fig. 1. Violation of the Law of Demeter (left) and modified design conforming to the Law (right)

We thus learn that it is indeed important to list the empirical relation systems beforehand. C&K share this opinion in [8] where they start off giving “empirical relation systems” called *viewpoints* for the metrics proposed. However, their viewpoints deal with the effects of each proposed metric on different (often external) attributes, e.g., for WMC: maintainability (Viewpoint 1) an reuse potential (Viewpoint 3); for DIT: predictability of behavior (Viewpoint 1) and reuse potential (Viewpoint 3); for NOC: reuse (Viewpoint 1) and correctness of abstraction (Viewpoint 2); for CBO: reuse (Viewpoint 1), maintainability (Viewpoint 2), and testability (Viewpoint 3); for RFC: testability (Viewpoint 1) and general complexity (Viewpoint 2); and for LCOM: encapsulation (Viewpoint 1) and correctness (Viewpoint 4). Discussing the effects of a metric on other attributes is certainly valuable, however, it can by no means replace the specification of a generally accepted empirical relation system for the attribute to be measured.

3. Issues of validation

Having established an empirical relation system, a metric M should then map the empirical relation system into an appropriate formal (or numerical) relation system, *preserving the semantics* of the empirical relation(s) observed. In other words, for every empirical relation $\angle$ and a corresponding formal relation $<$, the so-called *representation condition* $X \angle Y \Leftrightarrow M(X) < M(Y)$ must hold. The task of *validating* a software measure in the assessment sense is equivalent to demonstrating empirically that the representation condition is satisfied for the attribute being measured [11].

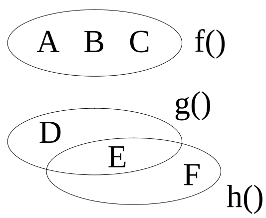
This validation of the measure can of course be complemented by other considerations, but can certainly not be totally replaced, as we will see in the following discussion of another very important attribute covered by C&K, namely, cohesion.

Cohesion is defined as an attribute of individual modules describing the extent to which the individual module components are needed to perform the same task [10]. In object-oriented systems, the term “module” in this definition is usually replaced by the term “class”. Seven different kinds of cohesion have been identified and ranked with respect to the strength of the resulting binding, ranging from (the weakest) *coincidental cohesion* which occurs when a class consists of a number of methods which do not seem to be related in any way to (the strongest and most desirable) *data cohesion*[4] which occurs

[4] The classical scale as proposed by Stevens et al. [21] has been slightly adapted for the object-oriented case by Budd [3].

when a class is used to implement a data abstraction, i.e., it defines internally a set of data values and exports methods that manipulate the data structure [3]. However, in order to correctly identify the cohesion type for a given class, a considerable amount of semantic information is needed which is very often not available for the (automated) process of metric collection. In this situation, C&K take a reasonable approach focussing on a simple notion of *similarity of methods* which only takes into account methods and their relationships to instance variables: Methods operating on a *common* set of instance variables are considered more similar than methods accessing *disjoint* sets of instance variables. The former contribute to high cohesion, while the latter reduce cohesion of the class, hinting at the possibility to split the class in two or more smaller classes.

To obtain an inverse measure of cohesion, C&K define *Lack of Cohesion in Methods* (LCOM) as the number of pairs of methods operating on disjoint sets of instance variables, reduced by the number of method pairs acting on at least one shared instance variable. For example, in class X in Fig. 2 below,




```
class X {
        int A, B, C, D, E, F;
        void f () { ... uses A, B, C ... }
        void g() { ... uses D, E ... }
        void h() { ... uses E, F ... }
};
```

Fig. 2. Example class with LCOM = 1

there are two pairs of methods accessing *no* common instance variables (<f, g>, <f, h>), while exactly one pair of methods shares variable E, namely, <g, h>. Therefore, LCOM is 2 - 1 = 1.

Unfortunately, this cohesion metric exhibits some anomalies with respect to the intuitive understanding of the attribute which will be explained below.

Consider the four designs presented in Fig. 3 where each Venn diagram represents a method by the set of instance variables it employs.

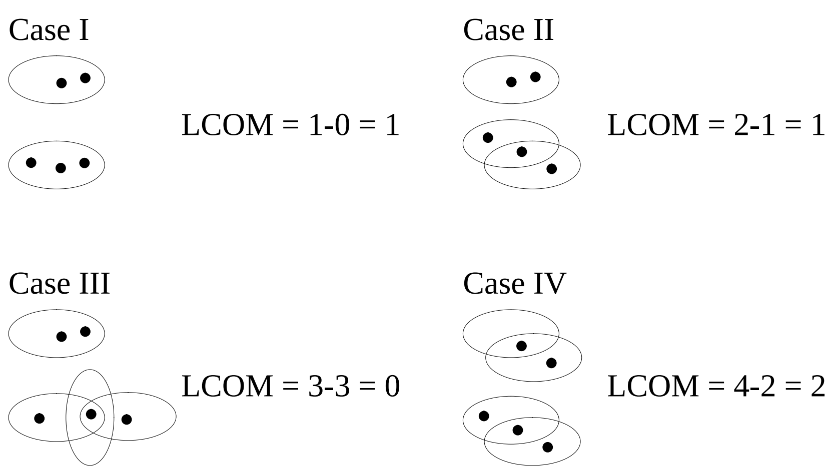


Fig. 3. Example LCOM computations

According to our intuition about cohesion, all of these cases are non-cohesive (which could be formalized in terms of an appropriate empirical relation): Without resorting to any semantic information about the classes, we are inclined to conclude that *all* classes should be broken up, despite the different

LCOM-values. Even according to C&K's viewpoint, "Lack of cohesion implies classes should probably be split into two or more subclasses" (page 489). According to the C&K metrics, the first two cases should be split (LCOM=1), case IV even more so (LCOM=2), while case III is considered structurally cohesive (LCOM=0) and therefore presumably would not be a candidate for splitting. Furthermore, it seems hard to explain why the addition of one method to an existing cluster in case I yielding case II should not change the cohesiveness, while performing the same operation again to case II can have conflicting effects: it reduces LCOM in case III and raises LCOM in case IV.

As another example, consider a (fictive) general class structure, where *n* methods are sequentially "linked" by shared instance variables as shown in Fig. 4.

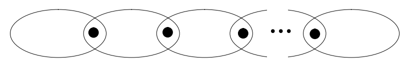


Fig. 4. Sequential cohesion

A short calculation for this special case yields

$$\mathrm{LCOM} = \left\lceil \binom{n}{2} - 2(n-1) \right\rceil^{+}$$

where $\lceil k \rceil^{+}$ equals *k*, if *k*>0 and 0 otherwise.

For *n*<5, LCOM is 0, well reflecting our intuitive view that all classes from this family are equally cohesive. However, for *n*=5, 6, 7, and 8, LCOM becomes 2, 5, 9, and 14, respectively, for the same structural pattern. On the other hand, if one argued that such a pattern can and should be split, thus supporting LCOM>0, the result LCOM=0 for *n*=3 and 4 is hard to explain.

It turns out that our intuition of this similarity-based notion of cohesion boils down to the approach of considering "clusters" of methods accessing common instance variables. It is interesting to note that in the earlier version of their metrics suite, C&K had probably the same idea in mind when they defined LCOM:

> *Consider a Class C_1 with methods M_1, M_2... , M_n. Let $\{I_i\}$ = set of instance variables used by method M_i. There are n such sets $\{I_1\}$, ... $\{I_n\}$.*
> *LCOM = The number of disjoint sets formed by the intersection of the n sets.* [7]

Although this older version - albeit the last sentence in this definition being somewhat ambiguous - in our interpretation does not exhibit the anomalies discussed above, C&K do not explain why they gave it up in [8].

Li and Henry attempted to rephrase the above definition in order to overcome the ambiguity as follows:

> *LCOM = number of disjoint sets of local methods; no two sets intersect; any two methods in the same set share at least one local instance variable; ranging from 0 to N; where N is a positive integer.* [15]

This version again supports our point of view. Taking this definition of LCOM, all our cases I-IV result in LCOM=2 while the pattern in Fig. 4 yields LCOM=1 for every *n*.

At this point, we propose a different, graph-theoretic formulation of Li and Henry's version of LCOM which is hopefully even more precise and also leads to a second (subordinate) metric which helps to differentiate among the ties in cases with LCOM=1 (for a more detailed discussion, see [12]).

Let X denote a class, I_X the set of instance variables of X, and M_X the set of its methods. Consider a simple, undirected graph $G_X(V, E)$ with $V = M_X$ and $E = \{<m, n> \in V \times V \mid \exists\, i \in I_X: (m \text{ accesses } i) \wedge (n \text{ accesses } i)\}$, i.e., exactly those vertices are connected which represent methods with at least one common instance variable. We can now define LCOM(X) as the number of connected components of G_X,that is, the number of method "clusters" operating on disjoint sets of instance variables. According to our interpretation of the definitions for LCOM in [7] and [15], the new formulation is equivalent.

In the cases where LCOM=1, there are still more and less cohesive classes possible. Especially for big classes, it might be desirable to refine the measure to tell the structural difference between the members of the set of classes with LCOM=1. For this purpose, let us consider the two extreme cases of connected graphs: The pattern in Fig. 4 which leads to a graph with $|E|=n-1$ represents the minimum cohesive case, while the a maximally cohesive design where all n methods access the same set of instance variables is mapped to the complete graph with $|E|=n\cdot(n-1)/2$. Thus, we can break many of the ties in the set of classes yielding LCOM=1 by considering the number of edges $|E|$: The more edges in G_X for a given method set $V=M_X$, the higher the cohesion of class X. For convenience, we map |E| into the interval [0, 1]:

$$C = 2\frac{|E| - (n-1)}{(n-1)\cdot(n-2)}$$

For classes with more than two methods, C can be used to discriminate among those cases where LCOM=1 as C gives us a measure of the deviation of a given graph from the minimal connective (that is, cohesive) case.

We conclude our discussion of LCOM with the observation that the above mentioned anomalies of LCOM have remained undiscovered in [8], in part because validation of the representation condition has been *substituted* by another rule set (Weyuker's axioms [23]). Put differently, we want to emphasize that any such set of validation criteria should only be employed in *addition* to the more fundamental representation condition.

As far as the applicability of Weyuker's properties is concerned, one should note that several defects have been identified in the past. C&K note this by citing formal criticisms by Cherniavsky and Smith [6], Fenton [10], and Zuse [27], and exclude three of the originally proposed properties from their list because they consider them trivially met by their metrics (properties 2 and 8) or not relevant for object-oriented design (Property 7). Moreover, in a more recent paper, Zuse has proved that Weyuker's axioms are contradictory within the representational theory of measurement ([28], cf. also [11]).

As a "philosophical" aside, we would like to point out a principle issue related to the application of Weyuker's axioms in C&K's paper: The validation of the three Weyuker's properties *monotonicity, nonequivalence of interaction,* and "*interaction increases complexity*" depends on the definition of a join operator ";" for software parts. For instance, the monotonicity criterion for metric $|\cdot|$ is defined as $\forall P \forall Q: |P| \leq |P\,;\,Q| \wedge |Q| \leq |P\,;\,Q|$ where P and Q denote software parts. It can be re-stated sloppily as "whenever you measure the complexity of a of a thing which is composed of two parts, the result must not be less than the measure applied to one of the parts in isolation". Our observation here is that the definition of operator "+" employed by C&K in place of Weyuker's ";" is not very well suited to Weyuker's rule set and might thus render the interpretation of validation results somewhat questionable (no matter whether they are positive or negative). In the remainder of this section, we explain our concerns with C&K's definition of operator "+".

Weyuker's "view of programs is that they are objects composed from simpler ... program bodies." ([24], pp. 1360-1361). She denotes a program built by sequential concatenation of parts P and Q by "P ; Q". In C&K (and below), the + operator is used to compose two objects. Thus, a 'whole' object P+Q is composed of two 'parts' P and Q. When one focusses on classes as building blocks in an object-oriented system (as C&K do), one needs a definition for $+: C \times C \rightarrow C$ (C denoting the set of classes) which combines two classes to a bigger thing which must be a class again. Given two classes A and B, there are several possible ways (cf. standard textbooks, e.g. [2][3][20][19][25]) to produce a new class AB "consisting" of both, A, and B:

a) Create AB which contains A and B as subobjects (aggregation). AB internally contains all properties of A and all properties of B, but does not present them to the outside unless a specific (additional) protocol is provided for that purpose.

b) Create AB by deriving B from A (i.e., leaving B as is except for deriving it from A). AB contains the bag of all properties of A and B, although A's instance variables which appear also in B are hidden in AB and A's methods which appear also in B are overridden in AB (but are nevertheless there).

c) Create AB by deriving a new class from both, A and B. AB again contains all properties of A and all of B, but some name conflict resolution mechanism must be used to avoid ambiguities.

d) Create AB by "merging" A and B in the C&K sense. In this case, AB receives the the *union of properties* (i.e., methods and instance variables) which means that common properties of A and B are reduced to a single appearance in AB (due to the union operator).

As we have shown, there are many ways to combine two classes into a single class. Thus, the most rigorous approach would be to check Weyuker's properties with respect to *all* of the above combination rules. However, if one restricts the validation process to only one of these possibilities, d) seems not the best choice, as it is not a very usual way of actually combining classes and it also involves the risk of inadvertently removing necessary properties when applying the union operator to seemingly "common" properties which happen to be homonyms denoting distinct concepts.

4. Conclusion

We have shown that the very interesting suggestions of C&K in the field of object-oriented design metrics can still be improved by adhering to some measure theoretic principles. Several authors have proposed a commonly agreed upon procedure to design useful measures for assessing (in contrast to predicting) attributes used in software development [1][10][11]. We have focussed on some of these steps in this paper, but would like to present an overview of the whole process from our point of view in Fig. 5. Revealing the possible error exits of this flowchart (which usually lead to iteration of the whole process or parts thereof) will help to analyze and improve previously proposed measures and may guide software scientists in the identification of new attributes and development of corresponding measures to avoid these branches in the first place.

As far as the concrete metrics discussed in this paper are concerned, we feel that it is most important that we exchange our intuitive understanding of the matter in order to arrive at a sound collection of measures. Thus, we agree with Fenton in [11]: "For many software attributes, we are still at the stage of having very crude empirical relation systems.". Although it will most probably take much more time and effort until we have arrived at our goal, we are certain, that the metrics community is on the right way.

Acknowledgements

We wish to thank the anonymous referees and the associate editor for their valuable suggestions as well as Norman Fenton for his comments on an earlier draft of this paper.

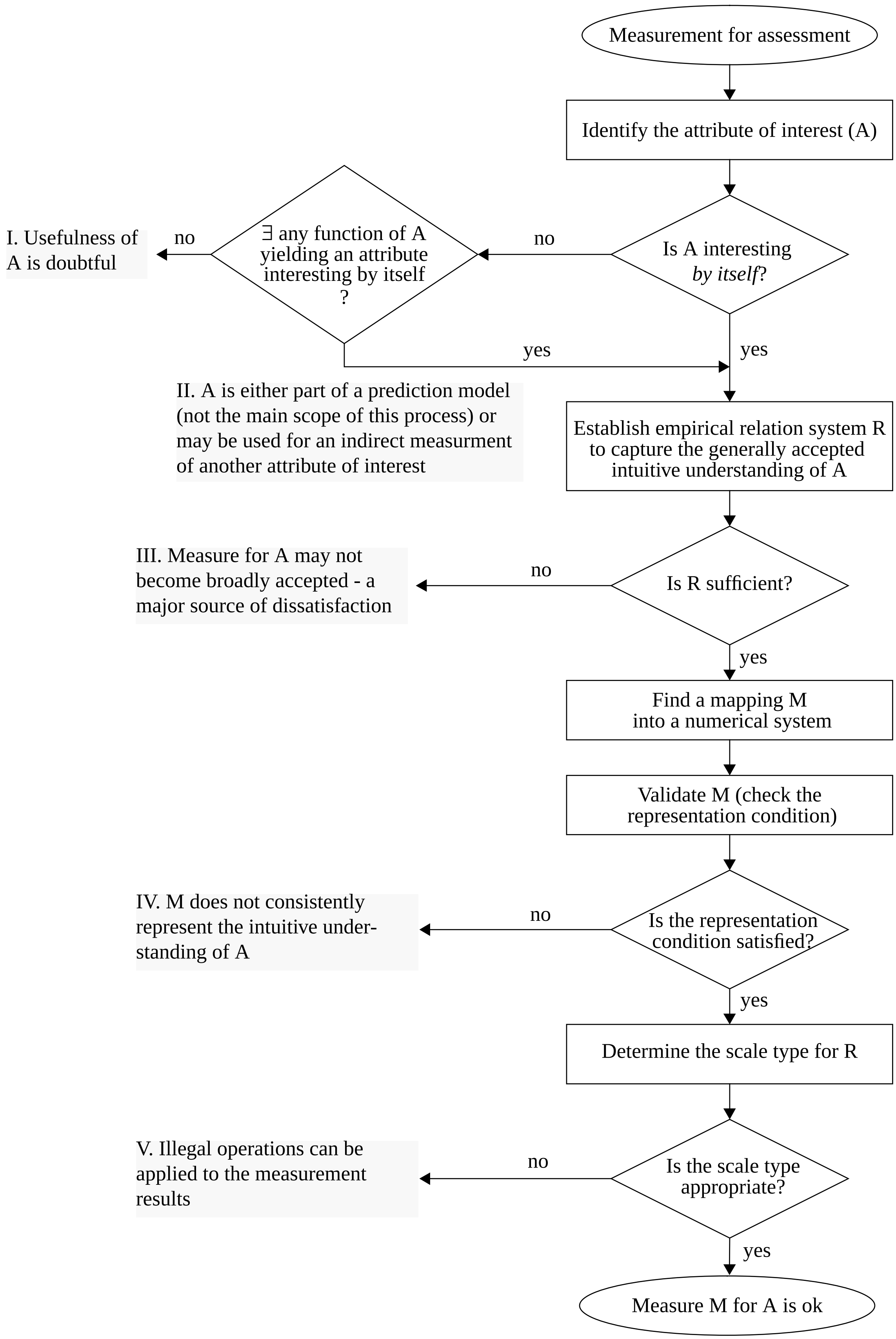


Fig. 5. Phases of measurement construction and possible mistakes

References

[1] A.L. Baker, J. M. Bieman, N. Fenton, D. A. Gustafson, and A. C. Melton, R. Whitty, “A Philosophy for Software Measurement”, *J. Systems Software* vol. 12, 1990, pp. 277-281.

[2] T. Bar-David, *Object-Oriented Design for C++*, Prentice Hall, 1993.

[3] T. A. Budd, *An Introduction to object-oriented Programming*. Reading: Addison Wesley, 1990.

[4] M. Bunge, *Treatise on basic philosophy: Ontology I: The furniture of the world*. Boston: Riedel, 1977.

[5] M. Bunge, *Treatise on basic philosophy: Ontology II: The world of systems*. Boston: Riedel, 1979.

[6] J. C. Cherniavsky and C. H. Smith, “On Weuker’s Axioms for Software Complexity Measures”, *IEEE Trans. Software Eng.*, vol 17 no. 6, pp. 636-638, 1991.

[7] S.R. Chidamber and C. F. Kemerer, “Towards a metrics suite for object-oriented design”, in *Proc. 6th OOPSLA Conference*, ACM 1991, pp. 197-211.

[8] S. R. Chidamber and C. F. Kemerer, “A Metrics Suite for Object-Oriented Design”, *IEEE Trans. Software Eng.*, vol. 20, no. 6, June 1994, pp. 476-493.

[9] N. I. Churcher and M. J. Shepperd, “Comments on ‘A Metrics Suite for Object-Oriented Design’”, *IEEE Trans. Software Eng.*, vol. 21, no. 3, March 1995, pp. 263-265.

[10] N. E. Fenton, *Software Metrics - A Rigorous Approach*. Chapman & Hall, 1992.

[11] N. E. Fenton, “Software Measurement: A necessary scientific basis”, *IEEE Trans. Software Eng.*, vol. 20, no. 3, March 1994, pp. 199-206.

[12] M. Hitz and B. Montazeri, “Measuring Coupling and Cohesion In Object-Oriented Systems”, in *Proc. Int. Symposium on Applied Corporate Computing (ISACC ‘95) , Oct. 25-27, 1995, Monterrey, Mexico*.

[13] I. Jacobson, *Object-Oriented Software Engineering - A Use Case Driven Approach*. Reading: Addison Wesley, 1992.

[14] J. P. LeJacq, “Semantic-Based Design Guidelines for Object-Oriented Programs”, in *JOOP, Focus on Analysis & Design*, 1991, pp. 86-97.

[15] W. Li and S. Henry, “Maintenance Metrics for the Object Oriented Paradigm”, in *Proc. 1st Int. Software Metrics Symp., Los Alamitos, CA, May 21-22 1993*, IEEE Comp. Soc. Press, 1993, 52-60.

[16] K. Lieberherr, I. Holland, and A. Riel, “Object-Oriented Programming: An Objective Sense of Style”, in *Proc. OOPSLA '88*, ACM 1988, pp. 323-334.

[17] K. Lieberherr and I. Holland, “Assuring Good Style for Object-Oriented Programs”, *IEEE Software*, September 1989, pp. 38-48.

[18] A. Macro and J. Buxton, *The Craft of Software Engineering*, Reading: Addison-Wesley, 1987.

[19] J. Martin and J. J. Odell, Object-Oriented Analysis & Design, Prentice Hall, 1992.

[20] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[21] W. Stevens, G. Myers, and L. Constantine, “Structured Design”, *IBM Systems Journal*, vol. 13, 1974, pp. 115-139.

[22] D. A. Troy and S. H. Zweben, “Measuring the Quality of Structured Designs”, *Journal of Systems and Software*, vol. 2, 1981.

[23] E. J. Weyuker, “Evaluating software complexity measures”, *IEEE Trans. Software Eng*., vol. 14, no. 9, September 1988, pp. 1357-1365.

[24] N. Wilde and R. Huitt, “Maintenance Support for Object-Oriented Programs”, *IEEE Trans. Software Eng*., vol. 18, no. 12, December 1992.

[25] R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.

[26] E. Yourdon and L. L. Constatine, *Structured Design. Fundamentals of a Discipline of Computer Program and System Design*. Yourdon Press Computing Series, Prentice-Hall, 1979.

[27] H. Zuse, “Properties of Software Measures”, *Software Quality J*, vol. 1, pp. 225-260, 1992.

[28] H. Zuse, “Support of Experimentation by Measurement Theory”, in *Experimental Software Engineering Issues* (LNCS vol. 706), H. D. Rombach, V. R. Basili, and R. W. Selby, eds., Springer Verlag, 1993, pp. 137-140.