Article can be accessed online at http://www.publishingindia.com

Impact, Effect and Relationship Amongst the Collection of Coupling Measures

Anjali Verma*, Bhupesh Kr. Dewangan**

Abstract

Several authors have tried to address the problem of impact, effect, and relationship amongst the collection of coupling measures by introducing frameworks to characterise different approaches to coupling and the relative strengths of these, although, on their own, none of the frameworks could be considered comprehensive. There are three existing and quite different frameworks for object-oriented coupling. First, Eder et al. identify three different types of relationships. These relationships, interaction relationships between methods, component relationships between classes, and inheritance between classes, are then used to derive different dimensions of coupling which are classified according to different strengths. Second, Hitz and Montazeri approach coupling by deriving two different types of coupling: object level coupling and class level coupling which are determined by the state of an object and the state of an object's implementation respectively. Again different strengths of coupling are proposed. And third, Briand et al. (1997) constitute coupling as interactions between classes. The strength of the coupling is determined by the type of the interaction, the relationship between the classes, and the interaction's locus of impact. As none of the frameworks has been used to characterise existing measures to the different dimensions of coupling identified, the negative aspects highlighted above are still very common ones.

Keywords: Coupling, Inheritance Coupling, Level of Coupling and Framework

Introduction

Object-oriented measurement has become an increasingly popular research area. This is substantiated by the fact that recently proposed in the literature are (i) several different frameworks for coupling and cohesion and (ii) a large number of different measures for object-oriented attributes such as coupling, cohesion, and inheritance. While this is to be welcomed, there are several negative aspects to the mainly ad-hoc manner in which object oriented measures are being developed. As neither a standard terminology nor formalism exists, many measures are expressed in an ambiguous manner which limits their use. This also makes it difficult to understand how different measures relate to one another. For example, there are many different decisions that have to be made when defining a coupling measure - these decisions have to be made with respect to the goal of the measure and by defining an empirical model based on hypotheses. Unfortunately, many measures proposed in the literature do not have the motivation behind these decisions documented, making it difficult to understand the underlying assumptions of the measure, e.g., it is often unclear what constitutes a coupling connection between two classes or what the strength of the coupling connection measured is compared to types of connections used by other coupling measures. As a result, it is also unclear what the potential uses of existing measures are and how different coupling measures could be used in a complementary manner

* MTech Scholar, Department of Computer Science & Engineering, Chhatrapati Shivaji Institute of Technology, Durg, Chhattisgarh, India. E-mail: anjali32324@gmail.com

** Sr. Assistant Professor, Department of Computer Science & Engineering, Chhatrapati Shivaji Institute of Technology, Durg, Chhattisgarh, India. E-mail: bhupesh.dewangan@gmail.com

to obtain a more detailed picture of the coupling in an object-oriented system

Literature Survey

During the last two decades, a lot of work has been done in the field of coupling measurement. Many researchers have worked in the field of coupling measurement of object oriented systems. Still there is no any standardisation and formalism in the field of coupling measurement. Every author has defined his measures on its own; no any measure is comprehensive. The frameworks defined by different authors are as below.

Frameworks for coupling in object-oriented systems have been proposed by Eder *et al.*, Hitz and Montazeri, and Briand *et al.* In each framework different types of class, method, and object coupling are identified.

Framework by Eder *et al.*

Eder *et al.* use the definition of coupling provided by Stevens *et al.* The authors identify the following types of relationships:

- Interaction relationships between methods. This type of relationship is caused by message passing.
- Component relationships. Each object has a unique identifier (the object identity). An object *o* may reference another object *p* using the identifier of object *p*. This introduces a component relationship between the classes of *o* and *p*.
- Inheritance relationships between classes.

From these relationships three dimensions of coupling are derived.

A. Interaction coupling: Two methods are interaction coupled if

(a) one method invokes the other, or

(b) they communicate via sharing of data.

Interaction coupling between method *m* implemented in class *c* and method *m´* implemented in class *c´* contributes to interaction coupling between the classes *c* and *c´*.

B. Component coupling: Two classes *c* and *c´* are component coupled, if *c´* is the type of either

(a) an attribute of *c*, or

(b) an input or output parameter of a method of *c*, or

(c) a local variable of a method of *c*, or

(d) an input or output parameter of a method invoked within a method of *c*.

C. Inheritance coupling: Two classes *c* and *c´* are inheritance coupled, if one class is an ancestor of the other. For each dimension of coupling, the authors identify different strengths of coupling (listed below from strongest to weakest).

Interaction Coupling

The definition of interaction coupling is most similar to the original definition of coupling. Therefore, Eder *et al.* use the types of coupling proposed by Myers and adapt these to object-oriented systems.

Content: Method accesses implementation of another. Implementation here means the non-public part of the class interface. In C++, for instance, a method *m* may be declared "friend" of a class *c*. Method *m* can then invoke private methods of class *c*. Access to implementation constitutes a breach of the information hiding principle and is considered the worst type of coupling.

Common: Methods communicate via unstructured, global, shared data space. The authors cannot give an example for an unstructured global shared data space, because there are no object-oriented languages which support them. Apparently, this type of coupling is only listed in order to be consistent with Myers' categories.

External: Methods communicate via structured, global, shared data space (e.g., a public attribute of a class). Eder *et al.* find that this is a violation of the "locality principle of good software design", without specifying any further what they mean by that.

Control: Methods communicate via parameter passing only, the called method controls the internal logic of the calling method. For instance, the called method may determine the future execution of the calling method. A change of the implementation of the called method will most likely affect the calling method (change dependencies).

Stamp: Methods communicate via parameter passing only, the called method does not need all of the data it receives. This constitutes an avoidable dependency

between methods. E.g., if the data structure of a parameter of a method is changed, possible effects of this change on the method have to be considered. If the parameter is unused, a change of the parameter's data structure will not have any effects. The effort spent to discover that the change has no effects can be saved by avoiding stamp coupling.

Data: Methods communicate via parameter passing only, the called method needs all the data it receives. This is the best type of coupling (besides no coupling at all), because it minimizes the change dependencies.

No direct coupling: No direct interaction coupling between two methods occurs. Eder *et al.* first consider only direct interaction coupling between two methods. Their definition is then expanded to indirect interaction coupling via transitive method invocations.

Component Coupling

There are four degrees of component coupling between classes (listed below from strongest to weakest).

Hidden: Component coupling does not manifest itself in code. For instance, if a class *c* contains a cascaded method invocation such as *a.m1().m2()*, the type of the object returned by *m2* need not be explicit in the interface or body of class *c*. It can be found in the interface of the class of the object returned by method *m1*. That is, in order to detect occurrences of hidden coupling where class *c* is involved, we also have to look at the interfaces of other classes.

Scattered: Component coupling manifests itself in the body of the class only (cases (c) and (d) in the above definition). Consequently, the body of the class has to be searched in order to detect occurrences of this type of coupling.

Specified: Component coupling manifests itself in the interface (cases (a) and (b)). It is sufficient to search the interface of the class for occurrences of this type of coupling.

Nil: No component coupling.

Inheritance Coupling

There are four degrees of inheritance coupling (listed below from strongest to weakest).

Modification: Inheriting class changes at least one inherited method in a manner that violates some predefined "good practice" rules. Eder *et al.* provide examples of such rules as "the signature of an inherited method *m* may only be changed by replacing the type of a parameter of *m*, say class *d*, with a descendent of class *d*", "an inherited method must not be deleted from the class interface", and "if a method is overridden, the overriding method must keep the same semantics as the overridden method". The predefined rules applied will, to a certain extent, depend on the used design methodology and programming language, but it should be obvious that such rules are subjective and not easily measured automatically. Modification coupling is the strongest type of inheritance coupling because information inherited from the parent class is modified or deleted in a manner which cannot be justified in the context of inheritance.

Two types of modification coupling exist.

a. ***Signature modification:*** Not only the implementation of at least one inherited method is changed, but the signature of the method is also changed.
b. ***Implementation modification:*** The implementation of at least one inherited method is changed. This degree of coupling is weaker than the previous type because the signature of the method is not changed.

Refinement: Inheriting class changes at least one inherited method but the change is made adhering to the predefined "good practice" rules. Refinement coupling is weaker than modification coupling because the inherited information is changed only according to the predefined rules. However, problems can still occur as a result of refinement coupling, e.g., changes to the signature of an inherited method will restrict the use of polymorphism even if the intended semantics of the method are not changed. Again, like modification coupling, there exist two different types of refinement coupling.

a. ***Signature refinement:*** not only the implementation of at least one inherited method is changed, but the signature of the method is also changed.
b. ***Implementation refinement:*** the implementation of at least one inherited method is changed. This degree of coupling is weaker than the previous type because the signature of the method is not changed.

Extension: Inheriting class changes neither the signature nor the body of any inherited method; only new methods and attributes are added.

Nil: No inheritance relationship between two classes.

Framework by Hitz and Montazeri

Hitz and Montazeri approach coupling by defining the state of an object (the value of its attributes at a given moment at run-time), and the state of an object's implementation (class interface and body at a given time in the development cycle). From these definitions, they derive two "levels" of coupling:

Class Level Coupling (CLC) CLC represents the coupling resulting from implementation dependencies between two classes in a system during the development lifecycle.

Object Level Coupling (OLC) OLC represents the coupling resulting from state dependencies between two objects during the run-time of a system.

According to Hitz and Montazeri, CLC is important when considering maintenance and change dependencies because changes in a server class may lead to changes in client classes. The authors also state that OLC is relevant for run-time oriented activities such as testing and debugging. For each of these levels of coupling, the authors identify a series of factors determining the strength of coupling.

Class Level Coupling

CLC can occur if a method of a class invokes a method or references an attribute of another class. In the following, let *cc* be the accessing class (client class), *sc* be the accessed class (server class).

The factors determining the strength of CLC between *cc* and *sc* are:

Stability of sc:

- ***sc* is stable:** interface or body of *sc* is unlikely to be changed (for instance due to changing requirements). Typically, basic types provided by the programming language, or classes imported from standard libraries are stable.
- ***sc* is unstable:** a class depending on an unstable server class is considered worse that depending on a stable class because a change to *sc* means potential change to *cc*. Typically, problem domain classes are unstable.

Two cases must be considered.

1. only the body of *sc* is likely to be changed.
2. the interface of *sc* may also be modified. This case is considered the more harmful modification.

Type of access:

- "Access to interface": *cc* invokes a method of *sc*.
- "Access to implementation": *cc* references an attribute of *sc*.

Access to implementation is considered stronger coupling as it constitutes a violation of the information hiding principle.

- **Scope of access:** Determines where *sc* is visible within the definition of *cc*. Within this scope, a change to *sc* may have an impact on *cc*. The larger the scope, the stronger the classes are coupled. The authors identify five cases which can be separated into two categories: (i) a reference to *sc* may occur in any method of *cc* and

(ii) a reference to *sc* can occur only through a particular method of *cc* (this becomes clear below). Category (i) comprises of three cases:

- *sc* is the type of an attribute of *cc*.
- *sc* is an ancestor of *cc*.
- *sc* is the type of a global variable.
- Category (ii) comprises of two cases:
- *sc* is the type of a local variable of a method of *cc*.
- *sc* is the type of a parameter of a method of *cc*.

Object Level Coupling

For the discussion of OLC (object level coupling), let *osc* be an object of type *sc*, *occ* an object of type *cc*. Three factors influence the strength of coupling between objects *osc* and *occ*:

- **Type of access:** *occ* accesses interface of *osc* or *occ* accesses implementation of *osc* (same distinction and implications for strengths of coupling as for CLC).
- **Scope of access:** The smaller the scope of access the weaker the coupling between the objects. For *occ* to be able to access *osc* and contribute to OLC object *osc* must be either (listed in increasing size of scope)

1. a parameter of a method of *occ*
2. a "non-native" part of *occ*, that is, *osc* is not an object inherited from a superclass of *occ* nor is it encapsulated (aggregation) within *occ* nor is it a local variable to one of *occ* methods
3. a global object.

- **Complexity of interface:** In the case that *occ* sends a message to *osc*, the number of parameters of the invoked method should be considered. The more parameters passed, the stronger the coupling between objects.

Framework by Briand *et al.*

An earlier approach to measure coupling in object-based systems such as those implemented in Ada is adapted to C++ by expanding it to include inheritance and friendship relations between classes. In contrast to the two previous frameworks, this framework focuses solely on coupling relationships available during the high level design phase. The motivation behind this decision is that eliminating design flaws and errors early before they can propagate to subsequent phases can save substantial amounts of money. As a result of the decision to focus on early design information, this framework concentrates on coupling caused by interactions that occur between classes. Three different facets are identified that determine the kind of interaction:

Type of Interaction

The framework determines the mechanism by which two classes are coupled. Different types of interaction are specified to determine if a particular type more accurately indicates fault likelihood.

Class-attribute interaction: There is a class-attribute interaction between classes *c* and *d*, if class *c* is the type of an attribute of class *d* (i.e., if aggregation occurs).

Class-method interaction: Let *md* be a method of class *d*. There is a class-method interaction between classes *c* and *d*, if

- class *c* is the type of a parameter of method *md*
- class *c* is the return type of method *md*.

Method-method interaction: Let *mc* be a method of class *c*, *md* be a method of class *d*. There is a method-method interaction between classes *c* and *d*, if

- *md* directly invokes *mc*
- *md* receives via parameter a pointer to *mc* thereby invoking *mc* indirectly.

Relationship

In C++, two classes *c* and *d* can have one of three basic relationships:

- **Inheritance:** Class *c* is an ancestor of class *d* or vice versa. This category is specified because the use of inheritance apparently contradicts the notion of minimising coupling and should be considered separately. Coad and Yourdon proposed a design principle which recommends high coupling between a class and its parents, and low coupling between classes not related via inheritance.
- **Friendship:** Class *c* declares class *d* its friend which grants class *d* access to non-public elements of *c*. This category is specified because it breaks encapsulation and thus violates the information-hiding principle.
- **Other:** No inheritance or friendship relationship between classes *c* and *d*.

Locus of Impact

If class *c* is involved in an interaction with another class, a distinction is made between

- **Export:** Class *c* is the server class in the interaction, and
- **Import:** Class *c* is the client class in the interaction.

The motivation for this distinction is to investigate whether direction is important for predicting the fault-proneness of a class. In the definition of this framework, Briand *et al.* deliberately assign no strengths to the different kinds of interactions they propose. The authors state such strengths should be derived from empirical validation which can then be used to define measures on an interval or ratio scale. Briand *et al.* pose several hypotheses regarding these facets of coupling and investigate these empirically with respect to prediction of fault prone classes.

Discussion and Comparison of Frameworks

A precise comparison of the frameworks shows there are differences in the manner in which coupling is addressed. One reason for this is the different objectives of the frameworks. For example, Briand *et al.* examined only early design information to investigate potential early quality indicators while other authors investigated information mainly available at low level design and implementation; hence differences are found in the mechanisms that constitute coupling. A second reason is that some of the issues dealt with by one set of authors are considered to be subjective and too difficult to measure automatically. For example, the stability of an individual class (addressed by Hitz & Montazeri) is not something which can be easily determined unless, say, all problem domain classes are classified as unstable. The following subsections discuss in detail the significant differences between the frameworks and what can be learned from these differences.

The Mechanisms that Constitute Coupling

Table 1 presents the mechanisms that constitute coupling according to each of the frameworks. Each row represents one mechanism; an "X" indicates that the mechanism is covered by the framework in the respective column. The mechanisms are numbered for reference purposes. There is little overlapping of the frameworks: method invocation is the only mechanism common to all three frameworks. Mechanisms 5 and 6 are common to the frameworks by Briand *et al*. and Eder *et al.* All other mechanisms are unique to one of the frameworks. Mechanism 9 (inheritance) is of a different nature than the other mechanisms. If two classes are connected via one of the mechanisms 1 to 8, then there is an actual client-server relationship between these classes: one class uses the other. If two classes are connected via mechanism 9, i.e., one class is the ancestor of the other, then there *can* (and probably should), but *need not* be any client-server relationship between the classes. The client-server relationship and the inter-dependencies this entails do not necessarily exist.

The coupling mechanisms differ in the development phase in which they become applicable. For instance, attribute references and method invocations (mechanisms 2 and 3) are completely known only after implementation. In contrast, aggregation is visible in the class interface and is typically available before implementation starts. Coupling mechanisms that are applicable early in the development process are particularly interesting. If, for instance, they help in identifying fault-prone classes, this information could be used to select classes which are to undergo formal verification, to allocate the best people to the most fault-prone parts of the design, or to select the optimal design from a series of design alternatives before these are implemented. However, the later the development phase, the more detailed the description of the system under development, the more detailed the analysis of coupling.

Strength of Coupling

The strength of coupling between two classes is determined by two aspects:

Table 1: Comparison of Existing Mechanisms that Constitute Coupling

#	Mechanism	Eder et al.	Hitz & Montazeri	Briand et al.
1	methods share data (public attributes etc.)	√		
2	method references attribute		√	
3	method invokes method	√	√	√
4	method receives pointer to method			√
5	class is type of a class' attribute (aggregation)	√		√
6	class is type of a method's parameter or return type	√		√
7	class is type of a method's local variable	√		
8	class is type of a parameter of a method invoked from within another method	√		
9	class is ancestor of another class	√		

- The frequency of connections between the classes, and
- The types of connections between the classes.

The first aspect, how to count the frequency of connections between classes, has to be ultimately resolved when defining the measures. This aspect has not been addressed by any framework probably because the information required to make this decision is available only after the source code is developed. Where the definitions of various proposed coupling measures are compared, it is shown that there are a number of different ways to count the frequency of connections between classes.

Different types of coupling have different strengths. Eder *et al.* and Hitz and Montazeri assign strengths to the types of coupling they identified by defining a partial order on the set of coupling types used in their frameworks. That is, for any two types of connections within each framework, they define if one is stronger than the other, if both have equal strength, or if their strengths are not comparable. It is important to note that the definition of such a partial order is to some degree subjective and requires empirical validation. Furthermore, the validity of a given order will clearly depend on the concrete measurement goal, i.e., different measurement goals can require different (partial) orders. For instance, assuming regression testing by means of structural testing based on control flow analysis is performed, we wish to estimate the effort to test a class based on the amount of import coupling of the class. We would be interested in method invocations because that influences the flow of control. We would not but be interested in references to attributes because these have no impact on the flow of control. If, on the other hand, we want to characterize the understandability of the class based on import coupling, direct references to attributes are likely to be equally important as method invocations.

Briand *et al.* define a set of measures which count for each interaction type of their framework the number of interactions a class has with other classes. Empirical validation is then conducted to evaluate their potential of identifying fault-prone classes. That is, strengths are empirically assigned to the different types of coupling.

Direction of Coupling

The framework by Briand *et al.* explicitly distinguishes import and export coupling. Consider two classes *c* and *d* being coupled through one of the mechanisms mentioned above. This introduces a client-server-relationship between the classes: the client class uses (imports services), the server class is being used (exports services). This distinction is important. A class which mainly imports services may be difficult to reuse in another context because it depends on many other classes. On the other hand, defects in a class which mainly exports services are particularly critical as they may propagate more easily to other parts of the system and are more difficult to isolate. We conclude that the direction of coupling measured directly influences the possible goals of measurement.

If two methods are coupled through "common" or "external" coupling according to the framework by Eder *et al.*, we cannot make a distinction between client and server so both methods would be clients. Note that with pure object-oriented languages this would not occur. If the global data space is a variable whose type is a class, this class could be considered the server.

Direct and Indirect Coupling

Eder *et al.* derive "indirect interaction relationships between methods" from "direct interaction relationships" using the transitive closure of direct interaction relationships. This idea can be applied to all kinds of coupling. If a class *c1* uses a class *c2*, which in turn uses a class *c3*, class *c1* is indirectly coupled to *c3*: a defect or modification in class *c3* may not only affect the directly coupled class *c2*, but also the indirectly coupled class *c1*. As an extreme case, consider a circular chain of coupled classes (class *ci* uses class *ci+1* for *i=1,2,...,n-1*, and class *cn* uses *c1*). Each class is directly coupled with two other classes (import and export coupling). However, each class in the chain indirectly uses and is being used by every other class.

Briand *et al.* based their framework on the work described. High-level design measures for coupling and cohesion in object-based systems were defined and validated with respect to their potential of identifying fault-prone modules. The coupling measures included measures for direct and indirect coupling. The measures for direct coupling were found to be useful predictors, those for indirect coupling, however, not. Therefore, Briand *et al.* did not include the distinction between direct and indirect coupling in their framework (because the framework has

primarily been defined to derive coupling measures for the identification of fault prone classes).

Stability of Server Class

This point is unique to the framework by Hitz and Montazeri. Using a stable class is better than using an unstable class, because modifications which could ripple through the system are less likely to occur. "Stability of the server class" could, for instance, be used to distinguish between classes imported from standard libraries (which usually are not being modified and thus are stable), and problem domain classes (which are unstable). Note that stability of a server class is a subjective concept which is difficult to measure automatically in a manner other than that suggested above.

Conclusion

To summarise, the following about coupling in object-oriented systems is noted:

a) There are different types of coupling among classes, methods, attributes

b) Classes and methods can be coupled more or less strongly, depending on

i. The type of connections between them

ii. The frequency of connections between them

c) A distinction can be made between import and export coupling (client-server relationships)

d) Both direct and indirect coupling may be relevant

e) The server class can be stable or unstable

f) The effect of inheritance on coupling has to be considered.

From this list we can see that there exists a variety of decisions to be made during the definition of a coupling measure. It is important that decisions are based on the intended application of the measure if the measure is to be useful. When no decision for a particular aspect can be made, all alternatives should be investigated. A second observation is that because the different aspects of coupling are independent of each other, a large number of coupling measures could be defined - this defines the problem space for coupling in object-oriented systems. In the following section, a review of object-oriented coupling measures in the software engineering literature is presented. Discussion of how existing measures address the different aspects of coupling takes place and insight is provided into how complete the overall problem space for coupling is covered by these measures.

References

Abreu, F., Goulão, M., & Esteves, R. (1995). *Toward the design quality evaluation of object-oriented software systems*. 5th International Conference on Software Quality, Austin, Texas, USA.

Arisholm, E. (2002). *Dynamic coupling measures for object-oriented software*. In Proceedings of IEEE Symposium on Software Metrics 8, 33-42.

Arisholm, E., Briand, L. C., & Føyen, A. (2004). *Dynamic coupling measurement for object-oriented software*. IEEE Transactions on Software Engineering, 30(8).

Basili, V., Briand, L., & Melo, W. (1995). *Measuring the impact of reuse on quality and productivity in object-oriented systems.* Technical Report, University of Maryland, Department of Computer Science, CSTR-3395.

Basili, V. R., Briand, L. C., & Melo, W. L. (1996). *A validation of object-oriented design metrics as quality indicators.* IEEE Transactions on Software Engineering, 22(10), 751-761.

Briand, L. C., Daly, J. W., & Wüst, J. (2002). *A unified framework for coupling measurement in object-oriented systems.* IEEE Transactions on Software Engineering, 25(1), 91-121.

Briand, L., Devanbu, P., & Melo, W. (1997). *An investigation into coupling measures for C++*. Technical Report ISERN 96-08. IEEE ICSE '97, Boston, USA, (to be published).

Briand, L., Emam, K. E., & Morasca, S. (1995). *Theoretical and empirical validation of software product measures.* Technical Report, Centre de Recherche Informatique de Montréal.

Briand, L., Morasca, S., & Basili, V. (1993). *Measuring and assessing maintainability at the end of high-level design.* IEEE Conference on Software Maintenance, Montreal, Canada.

Briand, L., Morasca, S., & Basili, V. (1994). *Defining and validating high-level design metrics.* Technical Report, University of Maryland, CS-TR 3301.

Briand, L., Morasca, S., & Basili, V. (1996). *Property-based software engineering measurement*. IEEE Transactions of Software Engineering, 22(1), 68-86.

Chidamber, S. R., & Kemerer, C. F. (1991). *Towards a metrics suite for object oriented design.* In A. Paepcke, (ed.) Proceedings of Conference on Object-Oriented Programming: Systems, Languages and Applications, 26(11), 197-211.

Chidamber, S. R., & Kemerer, C. F. (1994). *A metrics suite for object oriented design*. IEEE Transactions on Software Engineering, 20(6), 476-493.

Churcher, N. I., & Shepperd, M. J. (1995). *Comments on a metrics suite for object-oriented design*. IEEE Transactions on Software Engineering, 21(3), 263-265.

Harrison, R., Counsell, S., & Nithi, R. (2002). *Dynamic metrics for object oriented designs*. Proceedings of Software Metrics Symposium, 5, 150-157.

Poshyvanyk, D., & Marcus, A. (2006). *The conceptual coupling metrics for object-oriented systems*. Proceedings of the 22nd IEEE International Conference on Software Maintenance.

Yacoub, S. M., Ammar, H. H., & Robinson, T. (2002). *Dynamic metrics for object oriented designs*. Proceedings of Software Metrics Symposium, 6, 50-61.