# 2
# Design Modularity Measurement

Software metrics are a fundamental means to evaluate modularity of architectural and detailed design. This chapter provides an overview on how modularity is currently measured. This overview briefly describe representative existing metrics related either to architectural or detailed design. Our measurement approach also includes concern-driven design heuristic rules, which are mechanisms for supporting the interpretation for software measurements. Therefore, we also present here the state-of-art of conventional heuristic rules

## 2.1.
## Modularity Definition

Modularity has been playing a pervasive role in the context of software design. Many software engineering methods and techniques are based on the premise that a modular structure of software can improve its quality to some extent. According to a number of quality models, modularity is an internal quality attribute that influences external quality attributes (Fenton & Pfleeger, 1997), such as maintainability and reliability. It can be considered a fundamental engineering principle as it allows, among other things:

- to develop different parts of the same system by distinct people;
- to test systems in a simultaneous fashion;
- to substitute or repair defective parts of a system without affecting with other parts;
- to reuse existing parts in different contexts; and
- to restrict change propagation.

The IEEE Standard Glossary of Software Engineering Terminology (IEEE, 1990) defines modularity as the degree to which a system program is composed of discrete components such that a change to one component has minimal impact on other components. IEEE's definition is closely related to Booch's (1994) one, which states that modularity is the property of a system that had been decomposed into a set of cohesive and loosely coupled modules.

More recently, Baldwin & Clark (2000) have defined a theory which considers modularity as a key factor to innovation and market growth. This theory can be applied to different industries, including software development.

## 2.2.
## Modularity Attributes

Low coupling and high cohesion have been considered as the main drivers to achieve modular design (Myers, 1973; Stevens et al., 1979; Guezzi et al., 1991; Booch, 1994; Meyer, 1997; Fenton & Pfleeger, 1997; Pressman, 2000; Sommerville, 2000). Moreover, software engineers also regard narrow interfaces as an important driver for achieving modularity. A module interface should be as narrow as possible, since in a complex system a change in a module interface might result in changing many other modules (Booch, 1994; Meyer, 1997).

Although decomposing a system into independent modules is crucial, it is not sufficient to promote the benefits credited to a well modularized system. The criteria used to modularize the system might not be adequate to make the system concerns well localized in the modules. Even a system decomposed into weakly-coupled modules might suffer from poor changeability, comprehensibility and the like. For instance, consider a system that comprises several independent modules, which can even be compiled separately. If a functionality is spread over an expressive number of these modules, then the effort required to understand and change it might be high.

Actually, the concept of modularity applied to software development was first introduced by Parnas (1972), and his ideas are still considered extremely relevant. He claimed that the effectiveness of a modularization is dependent upon the criteria used in dividing the system into modules. He advocates that modularization is more about deciding what must be done by each independent module than just having independent modules. In particular, Parnas proposes that information hiding is a good criterion to be used. Therefore, based on these ideas, in this thesis, we consider modularity as:

- the degree to which the system concerns are well localized over the system modules, and
- the degree to which the system modules are cohesive, loosely-coupled, and have narrow interfaces.

Current quantitative assessment approaches of software design modularity usually rely on conventional abstractions such as module in order to undertake the measurements. Based on theses abstractions, they define and use metrics for quantifying attributes such as coupling between components, class cohesion, interface complexity, and so forth. Some metrics target architectural (or high-level) design, and are defined based on abstractions such as components, modules or packages (Section 2.3). Other metrics focus on detailed design modularity, and are rooted at abstractions such as classes and methods (Section 2.4).

All these metrics suffer from the limitations presented in Section 1.2. In spite of these limitations, these kinds of metrics still capture important aspects of software modularity. Therefore, we advocate that they should not be ruled out from the modularity assessment process and totally replaced by concern-driven metrics. Actually, concern-driven metrics are complementary to the conventional metrics. Therefore, our modularity assessment approach also includes conventional measurement. In particular, our architecture metrics suite (Chapter 4) encompasses metrics for coupling and interface complexity inspired on existing conventional metrics. The following sections briefly describe these conventional metrics.

## 2.3. Conventional Architecture Metrics

This section summarizes existing metrics for quantifying coupling and cohesion at high-level design. There are only few metrics that explicitly focus on high-level design abstractions. However, some detailed design metrics, later presented in Section 2.4, can also be applied to high-level design, if the design is decomposed in terms of object-oriented abstractions. In the following subsections, we describe the metrics which only focus on high-level design.

We do not discuss qualitative architectural assessment methods, such as SAAM and ATAM (Clements et al., 2002), because they traditionally focus on the architecture coverage of scenarios described in the requirements specification without a clear focus on quantitative modularity assessment. Differently, our work focuses on quantitatively assessing structural attributes of the architecture description which directly impact the architecture modularity.

### 2.3.1. Metrics by Martin

Martin (1997) defined two coupling metrics based on the abstraction of package, namely *afferent couplings* and *efferent couplings*. These metrics are defined on the package point of view. A package is an abstraction that encompasses a set of related classes. They count the number of dependencies that enter and leave a package. They are defined as follows:

- Afferent Couplings (Ca): this metric counts the number of classes outside a package that depend upon classes within this package.
- Efferent Couplings (Ce): this metric counts the number of classes inside a package that depend upon classes outside this package.

In order to illustrate the application of Martin's metrics consider the example of a high-level design in Figure 2. Consider the package in the center of the structure. There are four classes outside that package that depends upon the classes inside it. Thus, the value for the afferent couplings metric for that package is 4. Moreover, there are three classes outside the central package that are targets of relationships involving classes inside the central package. Therefore, the value for the efferent couplings metric is 3.

Martin claims that a package that depends upon a lot of other packages and has no other (or just few) packages depended upon is an instable package. First, the lack or small number of dependents makes the package easy to change. Besides, the packages that it depends upon may give it enough reason to change. On the other hand, a package that is depended upon by a high number of other packages, but itself depends upon only few packages is considered as stable by Martin. In this case, the package's dependents make it hard to change, and it has only few dependencies that might force it to change.

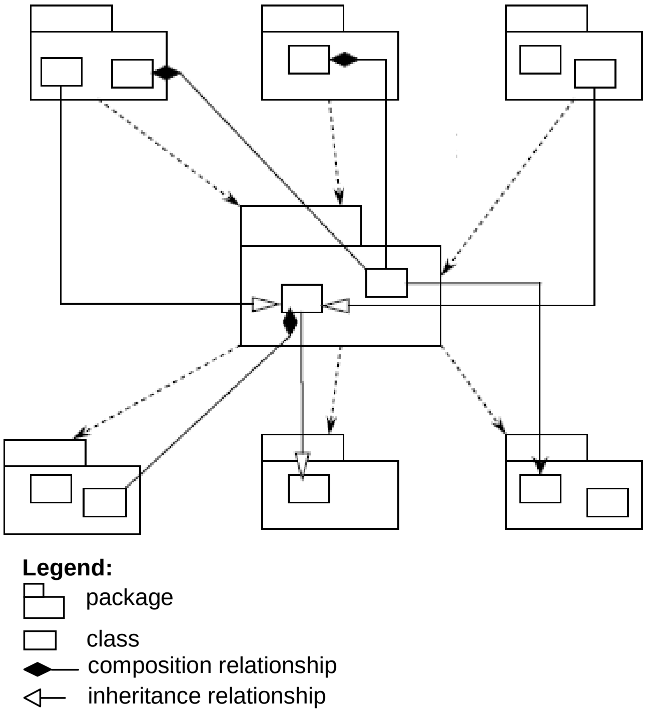

Figure 2: Design to illustrate Martin's coupling metrics

### 2.3.2. Metrics by Briand et al.

Briand et al (1993) also proposed a suite of architecture metrics for quantifying modularity-related attributes. The goal of their work was measuring coupling and cohesion in order to assess maintainability of high-level design. They aimed at measuring coupling and cohesion attributes with the purpose of predicting change difficulty. Thereby, they defined a suite of coupling and cohesion metrics based on the module abstraction. The concept of module used by them was taken from an object-oriented perspective (Ghezzi et al., 1991). Thus, in their approach a module is seen as a collection of routines, data and type definitions, i.e., a provider of services. A high-level design is a collection of module specifications related by "uses" or "is a component of" (Ghezzi et al., 1991) relationships.

In this context, Briand and colleagues defined modularity metrics based on the "uses" and "is a component of" relationships between modules. The metrics' definitions were presented in terms of interactions between data declared in the modules. However, the author clarified that, at the high-level design, there may not exist sufficient knowledge to precisely understand whether there will be an interaction between two data declarations in the final software system. Hence, their metrics only take into account data declaration interaction identified on the basis of the “uses” and “is component of” relationships. They proposed metrics for quantifying the import coupling and export coupling of a module. They defined *import coupling of a module* as the extent to which a module depends on imported external data declaration; and *export coupling of a module* as the extent to which a module’s internal data declaration affect the data declarations of the other modules in the system.

Their assumption to motivate the measurement of these two kinds of coupling is twofold. First, the more dependent a module on external data declarations, the more difficult it is to understand it in isolation. This is because the higher the import coupling, the more incomplete the local description of a module specification, the more spread the information necessary to isolate and understand a change. Second, they advocate that external coupling has a direct impact on understanding the effect of a change on the rest of the system. The larger the number of dependents of a module’s data declarations, the larger the likelihood of ripple effects when a change is implemented to that module.

Briand et al (1993) also proposed metrics for quantifying the cohesion of a module. They defined cohesion as the extent to which a module only contains data declarations which are conceptually related to each other. Thereby, their cohesion metrics are defined on the basis of the amount of interactions between the internal data declarations of a module. Their assumption is that, a high degree of cohesion is desirable because information relevant to a particular change within a module should not be scattered among irrelevant information. Therefore, data declarations which are not related to each other should be encapsulated to the extent possible into different modules.

### 2.3.3. Metrics by Lung & Kalaichelvan

Lung & Kalaichelvan (1999) presented an approach for quantitatively analyzing software architecture robustness. They define software architecture robustness as the degree of sensitivity of the architecture to the effects of change in stakeholder value parameters. In their context, stakeholders are customers, architectural evolution strategists and chief architects of the organizations. Examples of stakeholder value parameters include quality attributes like scalability and performance. They advocate that if an architecture is designed to accommodate many changes, then the architecture is robust. If some architectural elements that must be changed do not have much impact on the architecture or the downstream implementation, then the architecture is also robust.

To support a quantitative analysis of robustness, they presented a set of architecture metrics. Among other attributes, this set encompasses modularity-related metrics, more specifically, coupling between components metrics. The metrics presented by Lung & Kalaichelvan rely on the software architecture model defined by Perry & Wolf (1992). This model states that software architecture consists of three basic classes of architecture elements, namely processing element, data element and connecting element.

In this context, Lung & Kalaichelvan presented metrics for quantifying the fan-in and fan-out of processing elements. They did not present a precise definition of these metrics. However, we can grasp from the example given in their paper that the connecting elements are represented by arrows between the processing elements and data elements. The fan-in metric for a processing element counts the number of incoming arrows and the fan-out metric counts the number of outgoing arrows.

## 2.4.
## Conventional Detailed Design Metrics

Several detailed design metrics have been proposed in the past years. Since object-oriented design is the dominant design paradigm in current software engineering literature, we stick on studying and using object-oriented metrics in this work. Moreover, all the systems where we applied our metrics were designed in an object-oriented fashion. Object-oriented detailed design metrics are defined mainly on the basis of classes, objects, methods, attributes as well as association and inheritance relationships.

Object-oriented measurement has become an increasingly popular research area. This is substantiated by the fact that there is a growing number of different object-oriented measures for modularity-related attributes (Section 2.2) proposed in the literature. This section does not present an exhaustive list of these metrics. Rather, we describe here the ones that seem to be most theoretically validated and used in empirical studies. Besides, these metrics are supported by tools and seem to have a higher number of references in the literature. One of the most referenced suites of object-oriented metrics is the one proposed by Chidamber & Kemerer (1994). Their suite includes six metrics. In the following we present the definition of their metrics for coupling, cohesion and interface size. These metrics will be used later in one of our evaluation studies (Section 8.2).

### Weighted Methods per Class (WMC)

This metric is intended to relate to the notion of class complexity. For a class $c_1$ with methods $m_1$, $m_2$, …, $m_n$, weighted respectively with "complexity" $com_1$, $com_2$, …, $com_n$, the measure is calculated as $WMC = \sum_{i=1}^{n} com_i$ . Complexity is deliberately not defined more specifically in order to allow for the most general application of this metric. If all method complexities are considered to be unity, then $WMC = n$, the number of methods. In this case, this metric can be considered as a metric of interface size of a class. Interface size (or interface complexity) is one of the attributes that drives modularity (Meyer, 1997). We include interface size measures in our suite of metrics (Section 4.3.6).

### Coupling between Object Classes (CBO)

For a class *c*, this metric is defined to be the number of other classes to which *c* is coupled. CBO relates to the notion that a class is coupled to another class if methods of one class use methods or instance variables of another. Consider the predicate *uses*(*c, d*) defined in the following way: a class *c* uses a class *d* if a method implemented in class *c* references a method or an attribute implemented in class *d*. Let *C* be the set of classes in a system. Then CBO(*c*) is defined as: CBO(c) = |{d ∈ C – {c} | *uses*(*c*, *d*) ∨ *uses(*d, c*)*}|.

### Lack of Cohesion in Methods (LCOM)

This metric characterizes the lack of cohesion of a class by how closely its local methods are in terms of the local instance variables accessed in common. LCOM counts, for a class, the number of method pairs which access at least one instance variable in common, minus the number of method pairs that access no instance variable in common. Consider a class $C$ with $n$ methods $M1, M2, ..., Mn$. Let $Ii$ be the set of instance variables used by method $Mi$. Let $P = \{(Ii,Ij) \mid Ii \cap Ij = \varnothing \}$ and $Q = \{ (Ii, Ij) \mid Ii \cap Ij \neq \varnothing \}$. Then LCOM = $|P| - |Q|$, if $|P| \geq |Q|$, or LCOO = 0, if $|P| < |Q|$.

CBO (Coupling between Object Classes) is considered as a metric for coupling according to Briand and colleagues' theoretical frameworks for coupling (Briand et al., 1999) in object-oriented systems. CBO is based on method invocations and attribute references as the mechanisms that constitute coupling between two classes. Briand et al (1999) name coupling mechanisms as type of connection.

A plethora of other object-oriented metrics has been defined. Some of them are based on the same type of connections as CBO. For instance, the series of RFC (Response for a Class) metrics (Chidamber & Kemerer, 1991; Churcher & Shepperd, 1995), the MPC (Message Passing Coupling) metric (Li & Henry, 1993), and the ICP (Information-flow-based Coupling) family of metrics (Lee et al., 1995) are based solely on method invocation. As CBO, the COF (Coupling Factor) metric (Abreu et al., 1995) includes both method invocation and attributes references in its definition.

Other coupling metrics are derived from other types of connections. For instance, the DAC (Data Abstraction Coupling) and DAC' metrics take into account aggregation. In their unified framework for object-oriented coupling metrics, after reviewing a representative number of metrics, Briand et al (1999) summarized the possible types of connections in object-oriented coupling metrics. These types of connection are:

1. a class *d* is the type of an attribute of a class *c*;
2. a class *d* is the type of a parameter of a method, or the return type of a method of a class *c*;
3. a class *d* is the type of a local variable of a method of a class *c*;
4. a class *d* is the type of a parameter of a method invoked by a method of class *c*;
5. a method of class *d* references an attribute of class *c*;
6. a method of class *d* invokes a method of class *c*;

7. a class *d* has a high-level relationship (e.g. “uses”, consists-of”) with a class *c*.

In Chapter 5, we defined two new detailed design metrics which also include conventional connection between classes. Therefore, in their definition we specify which of the aforementioned types of connections are taken into account.

Cohesion is another important modularity-related attribute. Besides Chidamber & Kemerer’s LCOM metric, several other object-oriented cohesion metrics have been proposed in the literature. Briand et al (1998) also reviewed the existing cohesion metrics and distinguished two categories of types of connections. In the case of cohesion metrics, Briand et al refer to “type of connection” as the mechanisms that link elements within a class and thus make a class cohesive.

In the first category, they find measures focused on counting pairs of methods that use or do not use attributes in common. Chidamber and Kemerer’s LCOM metric falls into this category. In the second category, measures capture the extent to which pair of methods use attributes or invoke methods in common. We do not go further in details of conventional cohesion metrics because we do not include any of them in our approach. Our cohesion metrics are only based on the number of concerns in a component (Section 5.3.3).

Metrics for assessing coupling, cohesion, and interface size of aspect-oriented detailed design has been recently proposed (Ceccato & Tonella, 2004; Sant’Anna et al., 2003; Zhao, 2002, 2004; Zhao & Xu, 2004). Most of these metrics are extensions of object-oriented metrics. In order to increase the understandability of those metrics, we postpone the description of them until Chapter 3, where the concepts of aspect-oriented software development will be presented.

## 2.5. Design Heuristics Rules

Despite of being a powerful means to evaluate and control the quality of design, software metrics also pose some problems of their own: it is hard to provide a relevant interpretation for software measurements (Marinescu, 2004;

Lanza & Marinescu, 2006). The main problem is that the interpretation of individual measurements is too fine-grained if metrics are used in isolation. In most cases individual measurements do not provide relevant clues regarding the cause of a problem. A metric value may indicate an anomaly in the code but it leaves the engineer mostly clueless concerning the ultimate cause of the anomaly. Consequently, this has an important negative impact on the relevance of measurement results (Marinescu, 2004; Lanza & Marinescu, 2006).

In order to overcome this limitation of metrics, some researchers (Marinescu, 2004; Lanza & Marinescu, 2006) have recently proposed a mechanism for formulating metrics-based rules that capture deviations from good design principles and heuristics. Marinescu (2004) called this mechanism as *detection rules*. Sahraoui et al (2000) also used similar mechanism and called it just as *rules*. In this thesis, we call it as *design heuristic rules* (or simply heuristic rules).

Design heuristic rules aim at helping developers and maintainers detect and localize design problems in a system. A *design heuristic rule* is a composed logical condition, based on metrics, which detects design elements with specific problems. Using design heuristic rules an engineer can directly localize classes or methods affected by a particular design flaw (e.g., Shotgun Surgery (Fowler, 1999)), rather than having to infer the real design problem from a large set of abnormal metric values.

The use of metrics in the design heuristic rules is based on the mechanisms of filtering and composition. The filtering mechanism reduce the initial data set, so that only those values that present a special characteristic are retained. The purpose for doing that is to detect those design elements that have special properties captured by the metric (Marinescu, 2004; Lanza & Marinescu, 2006).

Marinescu (2004) grouped filters into two main categories: Absolute Filters and Relative Filters. *Absolute Filters* can be parameterized with a numerical value representing the threshold, e.g., HigherThan(20) and LowerThan (6) (Marinescu, 2004). *Relative Filters* delimit the filtered data set by a parameter that specifies the number of entities to be retrieved, rather than specifying the maximum (or minimum) value allowed in the result set. Thus, the values in the result set will be relative to the original set of data, e.g., TopValues(10) and BottomValues(5%) (Marinescu, 2004).

In addition to the filtering mechanism that supports the interpretation of individual metric results, the composition mechanism supports a correlated interpretation of multiple result sets. The composition mechanism is based on a set of operators that glue together different metrics in an articulated rule. (Marinescu, 2004; Lanza & Marinescu, 2006). Examples of these operators are the *and* and *or* logical operators.

In order to illustrate what a design heuristic rule is, we present here one of the rules proposed by Marinescu (2002). This rule aims at detecting a specific kind of modularity flaw, namely *Shotgun Surgery* bad smell (Fowler, 1999). The notion of bad smells was proposed by Kent Beck in Fowler's book (Fowler, 1999) to diagnose symptoms that may be indicative of something wrong in the design.

Marinescu's heuristic rule for detecting Shotgun Surgery is based on two conventional coupling metrics. This rule is defined as follows (Marinescu, 2002):

*Shotgun Surgery := ((CM, TopValues(20%)) and (CC, HigherThan(5))*

CM stands for the Changing Method metric (Marinescu, 2002, 2004; Lanza & Marinescu, 2006), which counts the number of distinct methods that access an attribute or call a method of the given class. CC stands for the Changing Classes metric (Marinescu, 2002, 2004; Lanza & Marinescu, 2006), which counts the number of classes that access an attribute or call a method of the given class. The Shotgun Surgery heuristic says that a class should not be the 20% with highest CM and should not have CC higher than 5, otherwise it represents an occurrence of the Shotgun Surgery bad smell.

To the best of our knowledge, all current design heuristics are based on conventional object-oriented metrics. Therefore, a common characteristic of these heuristics is that they are restricted to properties of modularity units explicitly defined in object-oriented programming languages, such as classes and methods. Hence, they automatically inherit the limitations of conventional metrics: they disregard the design driven concerns in design assessment processes. Chapter 5 illustrates the use of conventional design heuristic rules, discusses their limitations and proposes concern-sensitive design rules.