SOFTWARE QUALITY

A software product has to present several quality attributes. *Modularity* has been considered one of the most important software product quality criteria from an engineering point of view. For instance, in Ref. 1, modularity is cited as a criterion that can affect several software quality factors, such as efficiency, flexibility, interoperability, maintainability, reusability, and verifiability. A software product is considered *modular* if its components exhibit high cohesion and are weakly coupled (2). A module has high *cohesion* if all of its elements are related strongly. These elements, such as statements, procedures, or declarations, are used to achieve a common goal, which is the function of the module. On the other hand, *coupling* characterizes a module's relationship to other modules of the system. A coupling measure should gauge the level of the interdependence of two modules (e.g., module M calls a procedure provided by module N or accesses a variable declared by module N).

Weak module coupling is considered to be a desirable quality for modular programs. This belief stems from the supposition that a module will be easier to understand, modify, test, or reuse if it is weakly coupled with other modules. In addition, we believe that in that case an error in a module will propagate less into other modules of a system. Moreover, a weakly coupled module has a good chance to be less error-prone than a strongly coupled module.

Given the importance of coupling and cohesion for software quality, it is desirable to measure the cohesion and coupling of a software system. By doing so, we may be able to understand better the relationship between modularity (a software engineering design and implementation criterion) and software quality factors. Once we can measure the level of coupling and cohesion of a software system, we will be able to better characterize its quality, assess it with regard to other systems, and predict its product quality, for example with regard to maintenance costs and error-proneness.

The goal of this work is twofold. First, we are concerned with identifying the different forms coupling can take in a modular software system. As pointed out in Ref. 3, there are many different kinds of coupling. Each kind of coupling may have different effects on software quality. Second, we are engaged in measuring the different kinds of coupling and evaluating their effect on error-proneness (a software quality attribute).

RELATED WORK

Chidamber and Kemerer (4) have proposed a suite of object-oriented (OO) design metrics, called MOOSE metrics, which have been validated in Ref. 5. They provide a very simple coupling measure, called CBO. A class is coupled to another one if it uses its member functions and/or instance variables. CBO equals the number of classes to which a given class is coupled. Similarly to MOOSE, MOOD (6) includes a coupling measure, called the *coupling factor.* In MOOD, a class, A, is coupled with another one, B, if A sends a message to B. Both MOOSE and MOOD coupling measures are very simple and only take into account message exchange among classes.

Recently, Briand et al. (7) have defined a suite of coupling metrics for the design of OO systems. In this work a suite of 24 kinds of OO design coupling measures have been defined.

J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*. Copyright © 1999 John Wiley & Sons, Inc.

These coupling measures take into account different kinds of coupling that can exist in an OO-oriented design.

Regarding code coupling, in Ref. 3 eight different levels of coupling were proposed. For each coupling level, the shared data (parameters, global variables, etc.) are classified by the way they are used. In a more recent work, Offutt et al. (8) have extended the eight levels of coupling to twelve, offering a more detailed measure of code coupling. The coupling levels are defined between pairs of units, say P and Q. For each coupling level, the call/return parameters are classified by the way they are used. These uses are classified into computation uses (C uses), predicate uses (P uses), and indirect uses (I uses). We will detail these three kinds of uses in the next section.

Our work is inspired by Ref. 8. In addition, we have used the measurement framework proposed in Ref. 7. In fact, our work is complementary to that described in Ref. 7, in which OO design coupling measures were defined. Here, we are mainly concerned with coding coupling measures.

COUPLING OF MODULES

Before using the notions of software system, module, and modular system, let us introduce them. We adopt the basic definitions proposed by Briand et al. (9).

What Is a Modular System?

System. A system S is represented as a pair $\langle E, R\rangle$, where E represents the set of elements of S and R is a binary relation on E ($R \subseteq E \times E$) representing the relationships between S's elements.

Example. E represents the set of code statements and declarations and R the set of control flows from one statement to another.

Module. A module M of S is a pair $\langle E_M, R_M\rangle$, where E_M is a subset of E and R_M is a subset of $E_M \times E_M$ and of R.

Example. A module M could represent a code segment, a procedure, a set of such procedures packaged in the same file, or a class.

M's elements are connected to other system elements by incoming and outgoing relations InputR(M) and OutputR(M):

$$\begin{aligned}\mathrm{InputR}(M) &= \{(e_1, e_2) \in R | e_2 \in E_M \text{ and } e_1 \in E - E_M\}\\ &= \text{set of relationships from elements outside}\\ &\quad M \text{ to those inside } M\\ \mathrm{OututR}(M) &= \{(e_1, e_2) \in R | e_1 \in E_M \text{ and } e_2 \in E - E_M\}\\ &= \text{set of relationships from elements inside}\\ &\quad M \text{ to those outside } M\end{aligned}$$

Modular System. A modular system is a 3-tuple $S = \langle E, R, \mathrm{MC}\rangle$ where S is a system and MC a collection of S's modules such that

$$\forall e \in E\ (\exists M \in \mathrm{MC}\ (M = (E_M, R_M) \text{ and } e \in E_M))$$

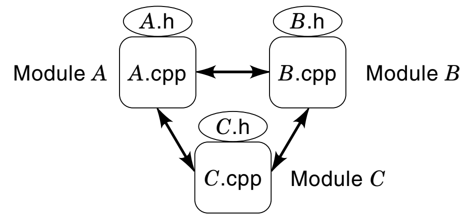

Figure 1. An example of modular system.

and

$$\begin{aligned}\forall M_i &= (E_{Mi}, R_{Mi}) \in \mathrm{MC},\ \forall M_j\\ &= (E_{Mj}, R_{Mj}) \in \mathrm{MC},\ E_{Mi} \cap E_{Mj} = \varnothing\end{aligned}$$

Example. Figure 1 shows the type of modular system we will consider in this empirical study.

In this context, coupling quantifies the strength of interconnection between modules of the same modular system.

Reference 9 states that a coupling measure must have certain properties—for example, be nonnegative, and null when there are no relationships among modules. Another important required property is that merging modules can only decrease coupling, so that we are encouraged to merge highly coupled modules in a single new module.

Before presenting in detail the set of identified *module* coupling levels, we specify the object of study. We consider a module as a collection of units, collected in a file and its associate header. A program *unit* is one or more contiguous program statements having a name by which other parts of the system can invoke it (e.g., procedure, function, method). We consider in our example that all modules are written in the C/C++ programming language.

A good software system should exhibit low coupling between units in different modules. Coupling increases the interconnections between the two units (and thus the two modules) and increases the probability that a fault in one unit will affect other connected units. In our context, we are interested in identifying possible interconnections between two units belonging to two different modules. We have to define different interconnection levels between two units m and n of two modules M and N. The architecture of such a system is illustrating in Fig. 1.

Identified Levels of a Module's Coupling

We distinguish between different kinds of module interconnections. Figure 2 shows this. If the modules are to be used

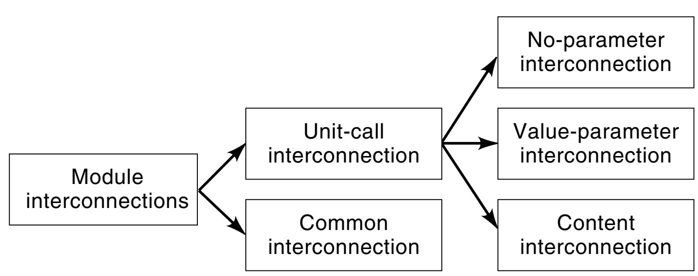

Figure 2. Module interconnection levels.

together in a useful way, there must be some external references, in which the code of one module refers to a location in another module. This reference may be to a data location defined in one module and used in another, as in *common interconnections.* On the other hand, it may be to the entry point of a unit (the callee) that appears in the code of one module and is called from another module unit (the *caller*). This is the case of *unit-call interconnection.* The distinction between different kind of modules interconnection is based on three criteria:

- The kind of information shared by interconnected modules (parameters or global areas)
- The type to which the shared information belongs (scalar, structure, class, etc.)
- What use is made of the shared information

In the context of the last criterion, Ref. 8 classifies uses into computational uses (*C use*), predicate uses (*P use*), and indirect uses (*I use*). A *C use* happens when a variable is used on the right side of an assignment statement or in an output statement. A *P use* occurs when a variable is used in a predicate statement. An *I use* occurs when a variable is used in an assignment to another variable, and this latter variable is then used in a predicate statement. Table 1 gives an illustration of this.

Table 1. Examples of Potential Uses of *x*

Type of Use	Examples
C use	`t=a*x*x−b*x;` `printf("%d\n",x);`
P use	`if ((x*x-4*a*c)>0)...`
I use	`t=a*x+b;...;` `while (t>0)...;`

Unit-Call Interconnection. This corresponds to the case where m calls n or n calls m, with or without passing parameters. In the case where m calls n, m is the *caller* and n the *callee.* We begin with the case where no parameters are transmitted from m to n.

No-Parameter Interconnection. Here m calls n or n calls m. There are no passing of parameters, references to common variables, or common references to external media. The number of occurrences of such interconnections, called NPI, is computed for each unit module, for each module, and then for the entire system.

Value–Parameter Interconnection. The modules M and N are connected through their respective units m and n. The caller m transmit parameters to the callee n, which uses them without modifying their values. The distinction between the following interconnection scenarios is based on two criteria: the type of the transmitted information and the use made of it. We wish to compute the number of occurrences of each scenario for each module unit, for each module, and for the whole system.

- *Scalar-Data Interconnection.* Some scalar variable in m is passed as an actual parameter to n, where it has a C use but no P use or I use.
- *Return-Data Interconnection.* m and n are connected by a return statement. The returned value has a C use.
- *Stamp-Data Interconnection.* A structure or a class object in m is passed as an actual parameter to n, and it has a C use but no P use or I use.
- *Scalar-Control Interconnection.* Some scalar variable in m is passed as an actual parameter to n, where it has a P use.
- *Return-Control Interconnection.* m and n are connected by a return statement. The returned value has a P use.
- *Stamp-Control Interconnection.* A structure or a class object in m is passed as an actual parameter to n, where it has a P use.
- *Scalar-Data–Control Interconnection.* Some scalar variable in m is passed as an actual parameter to n, where it has an I use but no P use.
- *Return-Data–Control Interconnection.* m and n are connected by a return statement. The returned value has an I use.
- *Stamp-Data–Control Interconnection.* A structure or a class object in m is passed as an actual parameter to n, where it has a I use but no P use.
- *Tramp Interconnection.* A variable x in m is passed to n; n passes x to another unit $p \in P$ without having accessed or changed the variable. The type of x may be scalar or structure/class.

Content Interconnection. This case occurs when the callee unit n of module N refers to and changes parameters passed by the caller unit m of module M. These parameters are passed by address (or reference). We identify eight kinds of such interconnections:

- *Scalar-Reference Data Interconnection.* The address of a scalar variable in m is passed as an actual parameter to n, where it has a C use.
- *Scalar-Reference Control Interconnection.* The address of a scalar variable in m is passed as an actual parameter to n, where it has a P use.
- *Scalar-Reference Data-Control Interconnection.* The address of a scalar variable in m is passed as an actual parameter to n, where it has a I use but no P use.
- *Scalar-Reference Modification Interconnection.* The address of a scalar variable in m is passed as an actual parameter to n, where it is modified.
- *Stamp-Reference Data Interconnection.* The address of a structure/class variable in m is passed as an actual parameter to n, where it has a C use.
- *Stamp-Reference Control Interconnection.* The address of a structure/class variable in m is passed as an actual parameter to n, where it has a P use.
- *Stamp-Reference Data-Control Interconnection.* The address of a structure/class variable in m is passed as an actual parameter to n, where it has a I use but no P use.
- *Stamp-Reference Modification Interconnection.* The address of a structure/class variable in m is passed as an actual parameter to n, where it is modified.

Common Interconnection. This corresponds to the case where two modules share same *global spaces.* Instead of communicating with one another by passing parameters, two modules access and eventually change information in a global area. We distinguish five interesting kinds of interconnection:

- *Global-Data Interconnection.* M and N share references to the same global variable. This latter is defined and used in N, and C-used in M. It is possible that this variable is not visible to the entire system.
- *Global-Control Interconnection.* M and N share references to the same global variable. This latter is defined and used in N, and P-used in M.
- *Global-Data–Control Interconnection.* M and N share references to the same global variable. This latter is defined and used in N, and I-used in M, but not P-used.
- *Global-Modification Interconnection.* M and N share references to the same global variable. This latter is defined and used in N, and accessed and modified in M.
- *Type Interconnection.* M and N share references to the same *user date type* (UDT). This UDT is defined and used in N, and used in M. This kind of interconnection includes what previous works called external-medium coupling (communication through a file, etc.).

The next subsection introduces the approach we have followed to measure the identified types of module coupling.

Measuring Module Coupling

In the previous subsection, we have listed and defined the identified types of module interconnection (MI). All these identified MIs are disjoint, so that if $\mathrm{MI}_i(M)$ is defined as

$$\mathrm{MI}_i(M) \subseteq \mathrm{InputR}(M) \cup \mathrm{OutputR}(M)$$

that is, the set of MI of type i in module M—then we have $\mathrm{MI}_i(M) \cap \mathrm{MI}j(M) = \emptyset\ \forall i, j$.

We will use a subset of $\mathrm{MI}_i(M)$: $\mathrm{MI}_i(m) \subseteq \mathrm{MI}_i(M)$ is the set of MIs of type i in unit m of M. On the other hand, we will specify, for each module interconnection type, the amounts of *importing* and *exporting* relative to the total amount of coupling. This quantifies the effect that one module's statements have on the statements of an interconnected module.

Figure 3 gives an example of a modular system with the computation of the import and export coupling for modules and units. The total coupling is computed as the sum of import and export coupling.

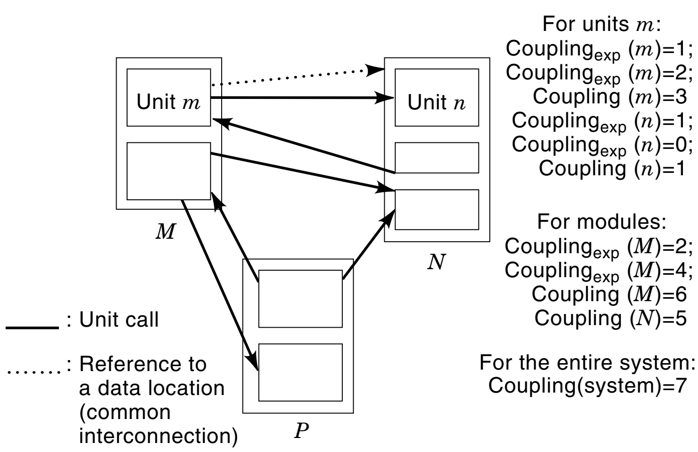

Figure 3. An example of coupling computations in modular systems.

For each module M and for the whole system, we define the *module interconnection measure* of type k, MIM_k as follows:

$$\begin{aligned}\mathrm{MIM}_k(M) &= \mathrm{MIM}_{k,\mathrm{imp}}(M) + \mathrm{MIM}_{k,\mathrm{exp}}(M)\\ &= \sum_{m@\text{ unit of } M} \mathrm{MIM}_{k,\mathrm{imp}}(m) + \sum_{m@\text{ unit of } M} \mathrm{MIM}_{k,\mathrm{exp}}(m)\end{aligned}$$

$$\mathrm{MIM}_k(S) = \sum_{M@\text{ module of } S} \mathrm{MIM}_{k,\mathrm{imp}}(M) = \sum_{M@\text{ module of } S} \mathrm{MIM}_{k,\mathrm{exp}}(M)$$

since the modules are disjoint. Here

$$\begin{aligned}k \in{} & \mathrm{Modules\_Coupling\_Type}\\ ={} & \{\mathrm{NPI, SDI, RDI, StDI, SCI, RCI, StCI, SDCI, RDCI,}\\ & \mathrm{StDCI, TI, SRDI, SRCI, SRDCI, SRMI, StRDI, StRCI,}\\ & \mathrm{StRDCI, StRMI, GDI, GCI, GDCI, GMI, TyI}\}\end{aligned}$$

EMPIRICAL VALIDATION OF COUPLING MEASURES

In this section, we address the empirical validation of the suite of measures introduced in the previous section. To do so, we use a product-metric validation (10). To better understand the relationship between code coupling and software quality, we have investigated the use of a machine learning algorithm to build characterization models (11).

Validation Data

In order to validate our suite of coupling measures, we have verified if these measures are useful to predict fault-prone classes. To do so, we have used the data from an open multiagent system development environment. This system has been developed and maintained since 1993. It contains 85 C++ modules/classes and approximately 47K source lines of C++ code (SLOC).

In this work, we have used: (1) the source code of the C++ classes, (2) data about these classes, (3) fault data. The fault data collected correspond to concrete manifestations of the errors found by the 50 beta testers of the system on versions 1.1a and 1.1. Version 1.1a was delivered in January 1997, and version 1.1 in November 1996.

The actual data for the suite of measures we have proposed were collected directly from the source code. The data preparation consisted in the extraction of seven types of facts. The resulted fact base was then exploited by a rule-based measuring system in order to infer for each module/class its associated MIM_k. It is important to note here that the measures were derived purely by static analysis. Only the classes that were developed by the development team were utilized. Classes reused from libraries or generated automatically by software tools were not used in this study, due to the obvious effects software reuse and code generators have on software quality (5).

Validation Strategy

To validate the OO design measures as quality indicators, we use a binary dependent variable aimed at capturing the fault-proneness of classes: did a fault occur in a class due to an operational failure? We used logistic regression (12) to analyze the relationships between our suite of measures and class fault-proneness. Logistic regression has already been successfully used to build software quality models and validate product software metrics, for example in Refs. 13, 5, and 7.

A multivariate logistic regression model is based on the following relationship equation (the univariate logistic regression model is a special case of this, where only one variable appears):

$$p(x_1, \ldots, x_n) = \frac{e^{a+\sum_i b_i x_i}}{1 + e^{a+\sum_i b_i x_i}}$$

where p is the probability that a fault occurred in a class during software operation and the X_i's are the coupling measures included as explanatory variables in the model [called covariates of the logistic regression equation (12)].

As in Refs. 5 and 7, we consider that an observation is the (non)detection of a fault in a C++ class. Each (non)detection is assumed to be an event independent of any other fault (non)detection. Each data vector in the data set describes an observation and has the following components: an event category (fault, no fault) and a set of measures characterizing either the class where the fault was detected or a class where no fault was detected. For each measure, we provide the following statistics:

- The coefficient b_i is the estimated regression coefficient. The larger the coefficient is in absolute value, the stronger the effect (positive or negative, according to the sign of the coefficient) of the explanatory variable on the probability p that a fault occurs in a class.
- $\Delta\psi$ is based on the notion of odds ratio and represents the reduction or increase in the odds ratio when the value of X increases by 1 unit (14). $\Delta\psi$ provides an insight into the effect of explanatory variables and is more interpretable than logistic regression coefficients.
- The statistical significance (*p-value*) provides an insight into the accuracy of the coefficient estimates. It tells the reader about the probability of the coefficient being different from zero by chance. The larger the level of significance, the larger the standard deviation of the estimated coefficients, and the less believable the calculated effect of the explanatory variables.

Results

Table 2 shows the descriptive statistics of measures extracted from the system under study. We can see that many measures have small variance in our data set. Indeed, six measures have no variance at all. As a consequence, at least in our data set, these measures are not likely to be useful predictors of class fault-proneness. This lack of variance may be explained by the fact that we are studying an OO product. OO products usually do not have the same kind of coupling we found in a procedural code.

With regard to G*xx* (i.e., GDI, GCI, GDCI, GMI) measures, the application we have investigated has very few global variables. In addition, global variables have not been used as parameters. It is important to realize that other applications may present different behavior. So, even we cannot validate these measures using this system, that does not imply that these measures will not add information when analyzing other systems. In addition, we are here dealing with an OO software system. According to OO design guidelines (15), global variables should not be used. Functions and procedures in a modular software system written in a procedural language, such as C or Pascal, may have a higher level of coupling than an OO one, due to exchange of global variables. Therefore, further investigation is necessary in order to assess the usefulness of these measures.

Table 2. Descriptive Statistics of Coupling Measures

Measure	Maximum	Minimum	Mean	Median	Std. Dev.
NPI	152	0	17.52	8	28.44
SDI	11	0	1	0	2.39
SCI	18	0	0	1.34	3.36
SDCI	10	0	0.29	0	1.37
RDI	88	0	9.64	4	13.83
RCI	49	0	7.24	3	10.7
RDCI	50	0	6.1	3	9.35
StDI	2	0	0.05	0	0.31
StCI	2	0	0.05	0	0.31
StDCI	2	0	0.048	0	0.308
TI	15	0	1.59	0	3.21
SRDI	19	0	0.61	0	2.32
SRCI	19	0	0.73	0	2.40
SRDCI	0	0	0	0	0
SRMI	0	0	0	0	0
StRDI	21	0	3.09	1	4.63
StRCI	21	0	2.69	1	4.75
StRDCI	11	0	0.939	0	2.19
StRMI	8	0	0.74	0	1.73
GDI	0	0	0	0	0
GCI	0	0	0	0	0
GDCI	0	0	0	0	0
GMI	0	0	0	0	0
TyI	40	0	10.65	8	8.73

With regard to SRDCI and SRMI and to the scalar variables used as parameters in module interchanges, it is comprehensible again that these measures present a very low variance. In OO software systems, in general, we transmit objects as parameters, not scalar variables as in procedural software systems. We can see in Table 2 that the measures St*xxx* (i.e., StDRI, StRCI, StRDCI, StRMI) have a higher variance than SRDCI and SRMI, since St*xxx* measures deal with structures and objects. Again, further investigation will be necessary to assess the importance of SRDCI and SRMI. However, we believe that SRDCI and SRMI will be exchangeable with St*xxx*: SRDCI and SRMI will be more useful when procedural software systems are studied, and Stxxx when OO software systems are.

Table 3 presents the measures that affect the predicted probability, in the C++ class, of (not) having a fault. Table 3 lists only the coupling measures that have a p value smaller than 0.05. These results have been obtained by univariate logistic regression analysis. Regarding $\Delta\psi$ (i.e., the effect of the measure on the probability of having a fault and the probabilityof not having one), the eight measures listed in Table 3 appear to have a considerable effect on p. For instance, there is an increase of 95.50% in the odds ratio ψ when NPI increases by one unit. Similarly, the measures RDI, RCI, RDCI, StRDI, and TyI have odds ratios greater than 80%. That

Table 3. Results from Univariate Logistic Regression

Measures	Coefficient	$\Delta\psi$	p value
NPI	−0.046	95.50%	0.007
RDI	−0.130	87.8%	<0.001
RCI	−0.092	91.21%	0.0003
RDCI	−0.142	86.76%	0.0003
SRDI	−1.195	30.27%	0.001
SRCI	−1.160	31.34%	0.001
StRDI	−0.106	89.94%	0.012
TyI	−0.198	82.04%	<0.001

Table 4. Two-Class Classification Performance Matrix

	Classified as		
Real value	No errors	Errors	Completeness
No errors	47	7	**87%**
Errors	11	20	**64%**
Correctness	**81%**	**74%**	
Overall correctness			78.82%
Overall misclassification			21.18%

means that individually these five coupling measures have pretty large effects on p.

FAULT-PRONE PREDICTIVE MODELS

In order to understand better the relationship between code coupling and software quality, we have built a characterization model, which can be used to easily assess error-prone modules/classes due to their level of coupling. The model-building technique that we used is a machine learning algorithm called C4.5 (16). C4.5 induces *classification models,* also called *decision trees,* from data. It is derived from the well-known ID3 algorithm (17). C4.5 works with a set of examples where each example has the same structure, consisting of a number of attribute–value pairs. One of these attributes represents the class of the example. The problem is to determine a decision tree that, on the basis of answers to questions about the nonclass attributes, correctly predicts the value of the class attribute. Usually the class attribute takes only the values {true, false}, or {success, failure}, or something equivalent.

The C4.5 algorithm partitions continuous attributes (in our case the coupling measures), finding the best threshold among the set of training cases to classify them on the dependent variable (in our case fault–no-fault classes). Classes with one or more faults have been classified as *fault,* and the other classes as *no-fault.* We chose this technique because the models are straightforward to build and are also easy to interpret. In addition, this class of modeling techniques has been used in the software engineering literature to build software quality predictive models (11), and therefore there already is some familiarity with it. Models built with C4.5 can be taken as complementary to the models built with logistic regression, mainly by software managers and software engineers who are not very familiar with statistical techniques.

To evaluate the class fault-proneness characterization model based on our coupling measures, we need criteria for evaluating the overall model accuracy. Evaluating model accuracy tells us how good the model is expected to be as a predictor. If the characterization model based on our suite of measures provides good accuracy, it means that our measures are useful to identify fault-prone classes. Three criteria for evaluating the accuracy of predictions are the predictive validity criterion, and measures of correctness and completeness.

Correctness is defined as the percentage of C++ classes (or modules) that were deemed fault-prone and were actually fault-prone. We want to maximize the correctness, because if it is low, then the model is identifying many C++ classes (or modules) as being fault-prone when they really are not fault-prone, which could lead to an overallocation of resources to verification and validation (i.e., to waste). *Completeness* is defined as the percentage of those faulty C++ classes (or modules) that were judged as fault-prone. We want to maximize completeness, because as completeness decreases, more C++ classes (or modules) that were fault-prone are misidentified as not fault-prone, which leads to a shortage of resources for verification and validation.

In order to calculate values for correctness and completeness, we used a V-fold cross-validation procedure (18). For each observation X in the sample, a model was developed based on the remaining observations (sample $-$ X). This model is then used to predict whether observation X will have errors or no errors. This validation procedure is commonly used when data sets are small.

Table 4 summarizes the quantitative results obtained with C4.5. As we can see, C4.5 presents good results in our experiment.

The model generated by C4.5 is composed of five rules (four explicit rules and a default rule). Figure 4 presents the induced rules. For example, rule 1 of the model can be read as: A class is erroneous *if* the no-parameters interconnection measure is greater than 16, *and* the return-data control interconnection measure is greater than 0, *and* the stamp-reference modification interconnection measure is less or equal to 3.

We have also investigated the usefulness of import and export coupling measures with regard to their capabilities to accurately predict/assess fault-prone classes by building two different predictive models: one using only importing measures, and another one using only exporting measures. Tables

Rule 0: SRCI $>$ 0
TYI $>$ 7
→ class error [89.9%]
Rule 1: NPI $>$ 16
RDCI $>$ 0
StRMI $\leq$ 3
→ class error [79.5%]
Rule 2: SDI $\leq$ 0
StRCI $>$ 2
StRMI $\leq$ 3
→ class error [64.5%]
Rule 3: NPI $\leq$ 16
SRCI $\leq$ 0
→ class no-error [80.2%]
Default class: no-error

Figure 4. The induced model.

Table 5. Fault-Prone Predictive Model Using Only Importing Coupling Measures

Real value	Classified as: No errors	Errors	Completeness
No errors	50	4	**92%**
Errors	8	23	**74%**
Correctness	**86%**	**85%**	
Overall correctness			85.88%
Overall misclassification			14.12%

Table 7. Fault-Prone Predictive Model Using Importing Coupling Measures and SLOC

Real value	Classified as: No errors	Errors	Completeness
No errors	50	4	**92%**
Errors	9	21	**70%**
Correctness	**84%**	**84%**	
Overall correctness			84.52%
Overall misclassification			15.48%

5 and 6 show these two models, respectively. We have found that the fault-prone predictive model based only on import coupling measures (Table 5) has proved to be more accurate than the predictive model shown in Tables 4 and 6. This result demonstrates that a class is most vulnerable to changes in its peers. Based on these results, we can say that one may use only importing coupling to predict fault-prone classes. If our only goal is to deal with fault-proneness, we can thus simplify the data-collecting programs and analysis, and by doing so we reduce by half the number of measures to be calculated.

However, exporting coupling measures might be useful to identify potential reusable classes. Classes that have a high level of exporting coupling measures should thus be further analyzed with a view to their inclusion in domain-specific component libraries. These classes, after a careful analysis, may be better documented to facilitate reuse. In this work we did not analyze classes reused from libraries, nor those generated automatically by programs. Further work is needed to verify if exporting coupling measures are useful to identify potential reusable classes.

COMPARISON BETWEEN OUR MEASURES AND EXISTING MEASURES

In this section we compare our measures with existing ones. As in Selby and Porter (19), we use a machine learning algorithm (e.g., C4.5) to select measures that are useful for predicting error-prone classes. If a measure is not selected, it was not useful as a predictor. We first compare our measures with the SLOC measure. This measure is extremely easy to calculate and has been used to build several quality models by different software organizations, e.g., NASA, SEL, and HP. SLOC represents the program size. In general, the larger a program is, the greater will be the number of defects. We also compare our measures with Chidamber–Kemmerer (CK) measures (4), a very well-known suite of OO design measures.

Table 6. Fault-Prone Predictive Model Using Only Exporting Coupling Measures

Real value	Classified as: No errors	Errors	Completeness
No errors	45	9	**83%**
Errors	17	14	**45%**
Correctness	**72%**	**60%**	
Overall correctness			69.41%
Overall misclassification			30.59%

CK propose a coupling measure called *coupling between object classes* (CBO), which states that class A is coupled to class B if A uses B's member functions and/or instance variables. CBO counts the classes to which a given class is coupled. Then, we analyze our measures with regard to C-FOOD (7), a suite of OO design measures specially conceived to identify coupling among classes. C-FOOD contains measures that can be applied at early phases of the product life cycle (high-level OO coupling design), as well as measures that can only be captured after the detailed design is accomplished (low-level OO coupling measures).

Table 7 shows the predictive model constructed by combining our importing measures and SLOC. It is important to note that SLOC was not selected. This means that SLOC was not a useful predictor of fault-prone classes when our importing measures are available. Note also that the results are almost the same as for the model presented in Table 5. The small differences are due to noise in the data introduced by SLOC. Although SLOC was not selected as a predictor of fault-prone classes, many papers in the literature show that this metric can be useful for characterizing, assessing, and predicting other attributes of software product quality. The results provided in this work, however, show that our importing coupling measures are better predictors of fault-prone classes than SLOC. Figure 5 and Table 7 summarize the obtained results. Further research is needed to verify if our metrics are also better predictors of other software quality attributes (e.g., software maintainability) than SLOC.

On comparing our measures with CK measures (4), we verified that CBO was the only CK measure selected by C4.5 algorithm. This means that CBO (the only CK measure that

Rule 0: SRCIimp > 0
→ class error [89.9%]
Rule 1: RDCIimp > 3
TIimp ≤ 0
→ class error [84.3%]
Rule 2: RDCIimp ≤ 3
SRCIimp ≤ 0
TYIimp > 4
→ class no-error [94.4%]
Rule 3: RDCIimp ≤ 1
SRCIimp ≤ 0
→ class no-error [84.8%]
Rule 4: TIimp > 0
SRCIimp ≤ 0
→ class no-error [79.5%]
Default class: error

Figure 5. Importing measures and SLOC-induced model.

Rule 0: RDIimp ≤ 2
SRCIimp ≤ 0
OMMEC ≤ 60
→ class no-error [96.3%]
Rule 1: RCIimp ≤ 5
SRCIimp ≤ 0
TYIimp > 6
AMMEC ≤ 5
→ class no-error [90.6%]
Rule 2: SRCIimp ≤ 0
TYIimp > 6
AMMIC ≤ 2
AMMEC ≤ 5
→ class no-error [89.9%]
Rule 3: SRCIimp > 0
→ class no-error [89.9%]
Rule 4: RDCIimp > 3
TYIimp ≤ 6
AMMEC ≤ 5
→ class err [87.1%]
Rule 5: OMMEC > 60
→ class error [79.4%]
Rule 6: RDIimp > 2
RDCIimp ≤ 3
OCMIC ≤ 10
→ class error [70.7%]
Default class: no-error

Figure 6. Model induced by importing and C-FOOD measures.

captures coupling) appears to be useful for identifying fault-prone classes due to code coupling. In fact, by the rules generated by C4.5, we have verified that in our data set, classes with CBO greater than 14 have a higher probability of faults. However, this number cannot be used for other data sets without a careful analysis. It is important to note that the results did not improve when we used CK measures. The predictive model for error-prone classes that only uses our importing measures is still the better one.

We have compared our measures with C-FOOD (see the results in Fig. 6 and Table 8). These results lead one to prefer the C-FOOD (7) low-level design coupling measures as useful predictors of fault-prone classes. They do not, however, lead one to reject C-FOOD high-level design measures, since these measures can be obtained earlier in the software product life cycle, whereas ours can only be obtained after the implementation stage. Based on these results, we can argue that C-FOOD measures seem to be more suitable for helping software managers decide which classes should be further inspected during the analysis/design phase, whereas ours are better to use during the validation phase to help software quality engineers decide which classes should be more carefully tested and/or recoded in order to reduce code coupling and, in consequence, improve software quality.

Table 8. Fault-Prone Predictive Model Using Importing Coupling Measures and C-FOOD

	Classified as		
Real value	No errors	Errors	Completeness
No errors	48	6	**88%**
Errors	10	21	**67%**
Correctness	**82%**	**77%**	
Overall correctness			81.18%
Overall misclassification			18.82%

From the results provided above we can draw the following conclusions:

- Our measures seem to be better predictors of fault-prone classes than are existing coupling measures. The use of other OO coupling measures did not improve the level of accuracy of the prediction model based only on our importing coupling measures.
- Our measures are better predictors of fault-prone classes than existing size measures such as SLOC. By the results presented above, our measures are not redundant with SLOC, which did not improve the model's predictive accuracy for class fault-proneness.
- Our measures can only be applied after the code is ready for analysis, whereas other OO coupling measures can be used at early phases of product life cycle. Therefore, our measures seem to be more suitable for use by software quality engineers at the maintenance/testing phase. OO design measures such C-FOOD and the CK measure can be used during analysis and design phases for helping software managers to select OO designs that have a low probability of faults. Given the fact that our measures act on the code, they can identify other forms of coupling that cannot be captured during analysis and design.
- The overall accuracy of the fault-proneness prediction models provided in this work is substantially higher than that of other models based on other measures.

CONCLUSION

The goals of this work are: (1) to identify the different forms coupling can take in a modular software system; (2) to measure these different forms of coupling via a mathematically sound set of coupling measures; (3) to validate these measures empirically by evaluating the effect of code coupling on error-proneness (a software quality attribute); (4) finally, to provide accurate predictive models based on these measures.

To validate our measures, we have used an industrial system. This system was implemented according to the OO paradigm. Some of our measures demonstrate a poor variance on this system and, in consequence, could not be empirically validated. We believe that our measures will behave differently when used on procedural-oriented software systems.

Despite the low variance of our data set, the results of our experimentation demonstrate that our measurse can predict fault-prone classes (an index of software reliability) with higher accuracy. Three predictive models have been generated. A subset of our measures proved to be quite accurate (92% completeness).

We intend to replicate this study using other data sets. In this work we have used an OO software system. Further studies are necessary to verify if we will continue to obtain such promising results using procedural code, written in C or Pascal.

ACKNOWLEDGMENTS

We wish to thank Mauricio A. De Almeida for his help in using the C4.5 algorithm. We are also grateful to E. Menif, Y.

Hemery, and H. Abassi for their help in collecting coupling data.

BIBLIOGRAPHY

1. I. Sommerville, *Software Engineering,* 4th ed., Reading, MA: Addison-Wesley, 1992.
2. L. L. Constantine and E. Yourdon, *Structured Design,* Englewood Cliffs, NJ: Prentice-Hall, 1979.
3. M. Page-Jones, *The Practical Guide to Structured Systems Design,* New York: Yourdon Press, 1980.
4. S. R. Chidamber and C. F. Kemerer, A metrics suite for object-oriented design, *IEEE Trans. Softw. Eng.,* **20**: 476–493, 1994.
5. V. Basili, L. Briand, and W. Melo, A validation of object-oriented design metrics as quality indicators, *IEEE Trans. Softw. Eng.,* **22**: 1996.
6. F. B. Abreu and R. Carapuxa, Candidate metrics for object-oriented software within a taxonomy framework, *J. Syst. Softw.,* **26** (1): 87–96, 1994.
7. L. Briand, P. Devanbu, and W. L. Melo, An investigation into coupling measures for C++, *Proc. 19th Int. Conf. Softw. Eng.,* Piscataway, NJ: IEEE Press, 1997.
8. A. J. Offutt, M. J. Harrold, and P. Kolte, A software metric system for module coupling, *J. Softw. Syst.,* **20** (3): 295–308, 1993.
9. L. C. Briand, S. Morasca, and V. R. Basili, Property-based software engineering measurement, *IEEE Trans. Softw. Eng.,* **22**: 68–85, 1996.
10. S. Benlarbi and W. L. Melo, *Measuring Polymorphism and Common Interconnections in OO Software Systems,* Tech. Rep. CRIM-97/11-84, Centre de Recherche Informatique de Montreal, Montreal, QC, Canada, 1997.
11. A. de Almeida, H. Lounis, and W. L. Melo, An investigation on the use of machine learned models for estimating software correctability, M, *Proc. 20th Int. Conf. Softw. Engi., Kyoto, Japan,* Piscataway, NJ: IEEE Press, 1998.
12. D. Hosmer and S. Lemeshow, *Applied Logistic Regression,* New York: Wiley-Interscience, 1989.
13. L. Briand, S. Morasca, and V. Basili, *Defining and validating high-level design metrics,* Tech. Rep. CS-TR-3301, Dept. Computer Science, Univ. Maryland, College Park, MD, 1994.
14. A. Agresti, *An Introduction to Categorical Data Analysis,* New York: Wiley, 1996.
15. G. Booch, *OO Analysis and Design with Applications,* 2nd ed., Menlo Park, CA: Benjamin Cummings, 1994.
16. J. R. Quinlan, *C4.5: Programs for Machine Learning,* San Mateo, CA: Morgan Kaufmann, 1993.
17. J. R. Quinlan, Discovering rules from large collections of examples: A case study, in D. Michie (ed.), *ES in the Micro-electronic Age,* Edinburgh: Edinburgh Univ. Press, 1979.
18. L. Breiman et al., *Classification and Regression Trees,* Belmont, CA: Wadsworth, 1984.
19. A. Porter and R. Selby, Empirically-guided software development using metric-based classification trees, *IEEE Softw.,* **7** (2): 46–54, 1990.

WALCÉLIO L. MELO
Oracle do Brasil and Universidade Católica de Brasilia

HAKIM LOUNÍS
HOUARI A. SAHRAOUI
Centre de Recherche Informatique de Montréal (CRIM)