/ 2604 p. 11

Measuring and Assessing Maintainability at the End of High Level Design

Lionel C. Briand, Sandro Morasca, Victor R. Basili
Computer Science Department and Institute for Advanced Computer Studies
University of Maryland, College Park, MD, 20742

Abstract

Software architecture appears to be one of the main factors affecting software maintainability. Therefore, in order to be able to predict and assess maintainability early in the development process we need to be able to measure the high-level design characteristics that affect the change process. To this end, we propose a measurement approach, which is based on precise assumptions derived from the change process, which is based on Object-Oriented Design principles and is partially language independent. We define metrics for cohesion, coupling, and visibility in order to capture the difficulty of isolating, understanding, designing and validating changes.

1 Introduction

It has been shown that system architecture has an heavy impact on maintainability [R90, S90]. Numerous studies have attempted to capture the high-level design characteristics affecting the ease of maintenance of a software system [HK84, R87, S90]. Research in the field of design metrics [G86, SB91, Z91, AE92] has often been conducted according to a strategy intended to produce generic metrics assumed to be applicable in a variety of contexts and to many problem domains. However, such an approach has forced researchers to work without a clear framework and a well-defined goal. This frequently led to some degree of fuzziness in the metric definitions, properties, and underlying concepts, making the use of the metric difficult, the interpretation hazardous, and the results of the various validation studies somewhat contradictory [IS88, K88]. Some attempts were made to constrain the context of application to a particular programming language in order to come up with precisely and unambiguously defined metrics[AE92]. In other cases, the application domain of those metrics was restricted, e.g., error-prone subprograms [SB91], maintainability [R87]. In all cases (with the exception of [AE92], where these issues were partially addressed), no precise link was made between the studied process (e.g., change process) and the metrics, no clear and precisely defined assumptions were made about the process itself, and metrics were not defined by taking into account the specific issue to be addressed (e.g., maintainability).

We intentionally place ourselves in a well-defined framework (Ada [DoD83] and OOD[BO87]) and intend to focus exclusively on the change process during acceptance testing and maintenance, i.e., the change process performed by personnel who did not develop the software. Thereby, we propose more precise and effective high-level design metrics based on well-defined and verifiable assumptions which are closely related to the specific change process model instantiated at the NASA Goddard Space Flight Center. Thus, the applicability of those metrics is precisely defined, their validation easier, and their predictive ability more accurate. However, we also attempt to separate Ada specific concepts from language independent concepts in order to identify the part of the approach that is reusable for other programming languages.

Our goals can be expressed by using Basili's G/Q/M template [BR88]:

> Analyze the *high-level design of a software system* for the purpose of *prediction* with respect to *change difficulty* from the point of view of the *testers and maintainers.*

> Analyze the *high-level design of a software system* for the purpose of *evaluation* with respect to *change difficulty* from the point of view of the *designers.*

From a modeling perspective, our long-term goal is to be able to build models that predict change difficulty for the maintenance process, which will provide an early evaluation of maintainability, thus allowing better architectural/design decisions. This paper first provides in Section 2 basic background information on the change process model and general definitions about the system constructs and the high-level design products. Section 3 presents the underlying concepts leading to the definitions of two basic metrics on top of which we define metrics for capturing module cohesion (Section 4), module coupling (Section 5), and coupling-based visibility control (Section

This work was supported in part by NASA grant NSG-5123, UMIACS, and NSF grant 01-5-24845

6). Finally, Section 7 summarizes the paper and presents the future directions of our research.

2 Background and Definitions

We first present the change process as perceived in our maintenance environment. Thus, we will be able to identify the various aspects of change difficulty (the *quality perspective* [BR88] of the goals of Section 1) and link our assumptions to this process so as to give a firm ground to our metrics. Then, we provide definitions for high-level design of a software system (the object of study of the goals of Section 1) and its basic constructs.

2.1 Change Process Model

In our environment of study (NASA Software Engineering Laboratory at the Goddard Space Flight Center), we view software maintenance as being composed of four primary phases, each encapsulating activities that may be performed concurrently, as shown in Figure 1.
There is a key milestone in the change process which is the decision of whether the change is going to be implemented or not. This is done based on a cost-benefit analysis after phase P1. The information necessary to this analysis is gathered during P1 and used for predicting the difficulty of designing, implementing and testing the change [BB92]. This information will encompass a description of the change itself and of the part of the system where the change is performed.

2.2 Object of Study

In the literature, there are two commonly accepted definitions of modules. The first one sees a module as a subprogram, and has been used in most of the design measurement publications [M77, CY79, HK84, R87, S90]. We choose the second category, which takes an object-oriented perspective, where a module is seen as a collection of routines, data and type definitions, i.e., a provider of computational services [BO87, G92].

Definition 1: Module.
A module is either a (possibly generic) subprogram, a (possibly generic) package, or a task. As such, a module comprises a specification and possibly a body.

Remark: Ada units vs. modules.
Compilation units are used in Ada for determining the compilation order and strategy. Instead, modules are defined here as Ada program units. We use the term module because it is a language independent concept. There are two kinds of Ada compilation units [DoD 83]: library units and secondary units. A library unit is either a package specification, a subprogram specification, or a whole subprogram which does not have a parent unit. Therefore, a library unit can be a module specification or a whole module. An Ada secondary unit is a unit with a parent unit and can only be a module body.

Definition 2: Data declaration.
A data declaration is either a type or an object (e.g., a constant, a variable, a formal parameter of a (possibly generic) subprogram or an entry, a generic formal object).

Definition 3: High-level Design product
The high-level design product is a collection of module specifications, either representing library units or belonging to secondary units, related by "uses" or "is a component of" [G92] relationships.

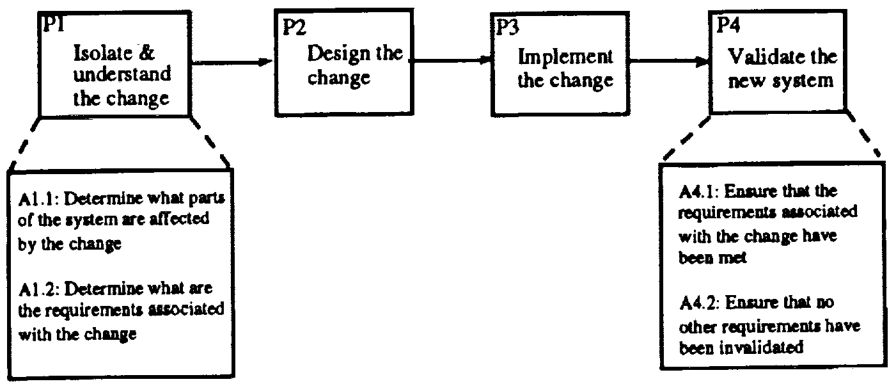
P1
Isolate & understand the change
P2
Design the change
P3
Implement the change
P4
Validate the new system
A1.1: Determine what parts of the system are affected by the change
A1.2: Determine what are the requirements associated with the change
A4.1: Ensure that the requirements associated with the change have been met
A4.2: Ensure that no other requirements have been invalidated

Figure 1. Change process

Since the remaining contents of units (e.g. local variables, algorithms) still remains to be determined in later design stages, our high-level design metrics will be mostly based on the information contained in module specifications. However, additional information to what is visible in the specifications may be available at the end of high-level design. For instance, given the specification of a module *m*, the designers have at least a rough idea of which objects declared in *m*'s and other modules' specification will be manipulated by a subprogram in *m*'s specification. It will be left to the person responsible for the metric program to decide whether or not it is worth collecting this kind of information, thus making the designer describe which global objects will be accessed by which subprograms or entries. For example, formatted comments might be a convenient way of conveying this information through module specifications and therefore of automating the collection of this type of information.

3 Interactions

We are looking for a primitive measure that links the change difficulty to the system design.
We therefore focus on the relationships that propagate side effects from data declarations to data declarations or subprograms when a change is performed. Those relationships will be called interactions and will be used to define metrics capturing cohesion and coupling within and between modules, respectively. Interactions linking subprograms to subprograms or data declarations will generally not be considered because they are encapsulated in module bodies and are therefore not detectable in our framework. However, these interactions are likely to be valuable although they will be rarely provided by the designer at the end of high-level design. For the sake of simplicity, we will not address this issue in the remainder of this paper. In all cases, these interactions will appear useful when looking at low-level design.

Definition 4: Data declaration-Data declaration (DD) Interaction.
A data declaration *A* DD-interacts with another data declaration *B* if a change in *A*'s declaration or use may cause the need for a change in *B*'s declaration or use .

The DD-interaction relationship is transitive. If *A* DD-interacts with *B*, and *B* DD-interacts with *C*, then a change in *A* may cause a change in *C*, i.e., *A* DD-interacts with *C*.

Data declarations can DD-interact with each other regardless of their location in the designed system. Therefore, the DD-interaction relationship can link data declarations belonging to the same module or to different modules.
By *DD-interactions(Dec_set1, Dec_set2)*, we will denote the number of DD-interactions from the set of data declarations *Dec_set1* to the set of data declarations *Dec_set2*.
At the end of high-level design, we may not have sufficient knowledge to understand with certainty whether there will be an interaction between two data declarations in the final software system, because we are not aware of all the DD-interactions present in the modules' bodies. On the basis of the information available from module specifications and their "uses" and "is a component of" relationships [G92], and from additional information provided by the designer, we can identify (1) the specification data declaration pairs that *are known* to DD-interact with each other, and (2) the specification data declaration pairs which *may* DD-interact with each other. We will say that there is an *actual DD-interaction* between data declaration pairs satisfying (1), and a *potential DD-interaction* between data declaration pairs satisfying (2). The latter kind of DD-interactions is only detectable by examining both specifications and bodies. Therefore, the set of actual DD-interactions is a subset of the set of potential DD-interactions.
The DD-interaction relationships can be defined in terms of the basic relationships between data declarations allowed by the language, which represent direct (i.e., not obtained by virtue of the transitivity of interaction relationships) DD-interactions. In Ada, data declaration *A* directly DD-interacts with data declaration *B* if *A* is used in *B*'s declaration or in a statement where *B* is assigned a value. As a consequence, as bodies are not available at high-level design time, we will only consider either the interactions detectable from the specifications or known by the designer.
DD-interactions provide a means to represent the relationships between individual data declarations. Yet, since procedures are not data declarations, DD-interactions *per se* are not able to capture the relationships between individual data declarations and subprograms, which are useful to understand whether data declarations and subprograms are related to each other and therefore should be encapsulated into the same module (see Section 4 on module cohesion).

Definition 5: Data declaration-Subprogram (DS) Interaction.
A data declaration DS-interacts with a subprogram if it DD-interacts with at least one of its data declarations.

Whenever a data declaration DD-interacts with *at least one* of the data declarations contained in a subprogram specification, the DS-interaction relationship between the data declaration and the subprogram can be detected by examining the high-level design. For instance, from the code fragment in Figure 2, it is apparent that both type *T1* and object *OBJECT11* DS-interact with procedure *P11*, since they both DD-interact with parameter *PAR11*, one of procedure *P11*'s specification data declarations.

```
package Pk1 is
  ...
  type T1 is ...;
  OBJECT11, OBJECT12: T1;
  procedure P11(PAR11: in T1:=OBJECT11);
  ...
  package Pk2 is
        ...
        OBJECT13: T1;
        type T2 is array (1..100) of T1;
        OBJECT21: T2;
        procedure P21(PAR21: in out T2);
        ...
  end Pk2;

  task Tk is
        entry E1(PAR12: in out T1);
        entry E2(PAR22: in out T2);
  end Tk;
  ...
  OBJECT22: Pk2.T2;
  ...
end Pk1;
```

Figure 2. Program fragment

On the other hand, there may be DS-interactions that are not detectable only on the basis of the Ada code representing the high-level design, since they are due to DS-interactions occuring in subprogram bodies. For instance, from the code fragment above, we cannot tell whether *OBJECT12* DS-interacts (as a global variable) with procedure *P11*. The designers may very likely be able to supply this additional piece of information. More specifically, the designers can answer in three different ways:

(1) *OBJECT12* DS-interact with *P11*
(2) *OBJECT12* does not DS-interact with *P11*
(3) the information they have is not sufficient

It is worth saying that answers of kind (2) provide valuable, though negative, information on the DS-interaction present in a system.

Remark:
Definition 5 states that DS-interaction is a relationship between data declarations and a subprogram, which is a specific kind of module. Since we are interested in the interactions between data declarations and algorithms, we did not provide a more comprehensive definition also accounting for the relationships between a data declaration and a package or a task, which are the other possible kinds of module. As a matter of fact,

- packages are a means for grouping/encapsulating data declarations and subprograms (and possibly tasks and other packages). Therefore, we will not examine the relationships between a data declaration and a package as a whole.

- tasks are defined in terms of their entries, i.e., they can be seen as a collection of entries, which we will see as a particular kind of subprograms. Therefore, we will not examine the relationships between a data declaration and a task as a whole.

For graphical convenience, both sets of interaction relationships will be represented by directed graphs, the *DD-interaction graph*, and the *DS-interaction graph*, respectively. In both graphs (see Figures 3 and 4, which respectively represent DD- and DS-interaction graphs for the code fragment of Figure 2), data declarations are represented by rounded nodes, subprograms by thick lined boxes, and packages and tasks by thin lined boxes. Solid arcs represent interactions that can be known by either inspecting the high-level design or collecting information from the designers, dashed arcs represent those interactions that are not detectable from the high-level design and that will not occur in the body, according to the designers' opinion. (For simplicity's sake, in Figure 3 we only represent *direct* DD-interactions.) For instance, the existence of an DD-interaction between object *OBJECT12* and *PAR11* and the lack of interaction between *OBJECT13* and *PAR21* have been signaled by the designer. Since this information may improve significantly the accuracy of the count of DS-interactions and is in many cases known by the designers, we strongly recommend that the reader pay attention to this issue.

Our approach to design measurement and evaluation will be based on the above definitions and will be guided by the general principle that system architecture should have low average module coupling and high average cohesion. This is assumed to improve the capability of a system to be decomposed in highly independent and easy to understand pieces. Cohesion captures the extent to which the data declarations and subprograms that interact are grouped within the same modules, whereas coupling captures their dispersion by looking at module dependencies and exports. These issues are addressed in the next sections.

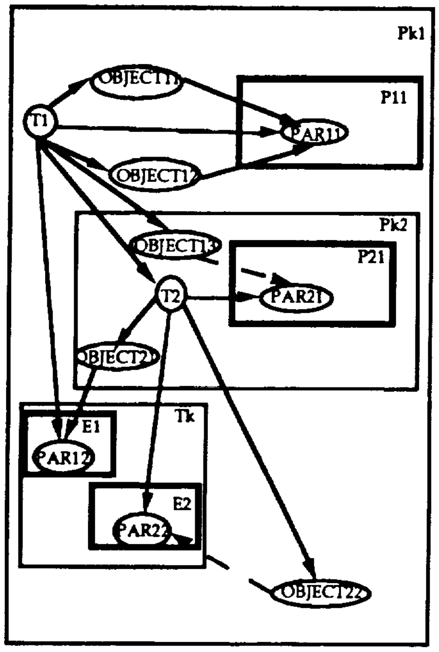

Figure 3. DD-interaction graph for the program fragment in Figure 2

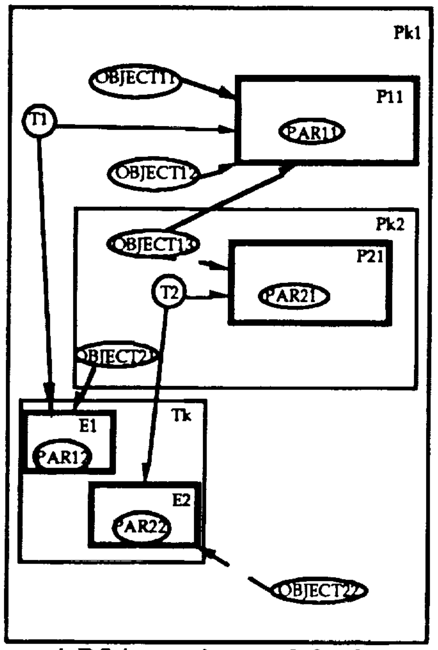

Figure 4. DS-interaction graph for the program fragment in Figure 2

4 Module Cohesion

It is generally acknowledged that a high degree of cohesion is a desirable property of a module. Here, after a general definition for cohesion, we provide assumptions to restrict it to our specific viewpoint—change. This allows the definition of change-oriented cohesion metrics which are also based on our OOD definition of module.

4.1 Definitions

Definition 6: Cohesion (CH)
Cohesion is the extent to which a module only contains data declarations and subprograms which are conceptually related to each other.

Assumption A-CH:
From our "change process" viewpoint, a high degree of cohesion is desirable because information relevant to a particular change within a module should not be scattered among irrelevant information. Data declarations and subprograms which are not related to each other should be encapsulated to the extent possible into different modules. We believe that this issue is especially important for activity *A1.2* (see Figure 1) where the change requirements have to be understood.

4.2 Cohesive Interactions

Since we place ourselves at the end of high-level design and we want to look at the set of services provided by a module, we are interested in evaluating how tight are the relationships between the data declarations declared within a module specification, and between the data declarations and the subprograms declared there. We will capture this by means of cohesive interactions.

Definition 6: Cohesive Interaction.
The set of cohesive interactions in a module is the union of the sets of DS-interactions and DD-interactions, with the exception of those DD-interactions between a data declaration and a subprogram formal parameter.

We do not consider the DD-interactions linking a data declaration to a subprogram parameter as relevant to cohesion, since they are already accounted for by DS-interactions and we are interested in evaluating the degree of cohesion between data declarations (data), and procedures (algorithms) seen as a whole.

Remark.
It is worth reminding the reader that those relationships that cannot be detected by inspecting the specifications, i.e., global variables interacting with subprogram bodies, can actually be quite relevant to cohesion evaluation, because they often represent the connections between an object and

the subprograms that access it; such connections are the relationships that make an abstract object cohesive.

4.3 Cohesion Metrics

Based upon the above definition of cohesive interactions, we define a cohesion metric that satisfies the following two properties.

Property 1: Normalization.
Given a module m, the metric $cohesion(m)$ belongs to the interval [0,1].

Normalization allows meaningful comparisons between the cohesions of different modules, since they all belong to the same interval.

Property 2: Monotonicity.
Let m_1 be a module and CI_1 its set of cohesive interactions. If m_2 is a modified version of m_1 with one more cohesive interaction so that CI_2 includes CI_1, then $cohesion(m_2) \geq cohesion(m_1)$.

Since there is uncertainty on the DD- and DS-interactions present in a module, due to the incompleteness of the information that can be collected from the specifications and the designers, we define not only a metric but the boundaries of an uncertainty interval.

Definition 7: Ratios of Cohesive Interactions.

Neutral Ratio of Cohesive Interactions (NRCI)*:*
All unknown CIs are not taken into account

NRCI=#knownCIs/(#potentialCIs-#unknownCIs)

Pessimistic Ratio of Cohesive Interactions (PRCI)*:*
All unknown CIs are considered as if they where known not to be actual interactions.

PRCI = #knownCIs/#potentialCIs

Optimistic Ratio of Cohesive Interactions (ORCI)*:*
All unknown CIs are considered as if they where known to be actual interactions

ORCI=(#knownCIs #unknownCIs)/#potentialCIs

If PRCI, NRCI, and ORCI are all not undefined, it can be shown that

$$PRCI \leq NRCI \leq ORCI$$

Figure 5 shows representative and interesting examples of module cohesion computation. Each thin lined box represents a module specification. T's, O's, and SP's will characterize types, objects and subprograms, respectively. We did not represent procedure parameters, since they do not belong to any cohesive interaction, nor packages nor tasks, since they are inessential to our discussion. However, we represented all direct and transitive interactions.

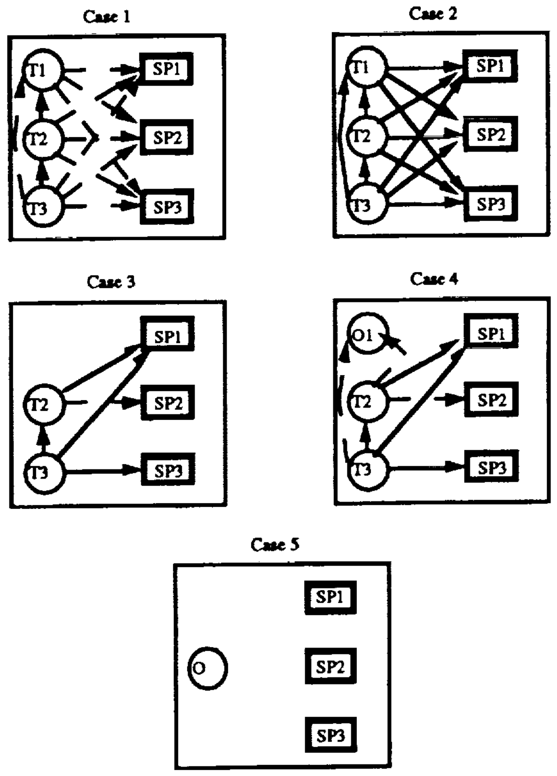

Figure 5: Cohesion examples

Case 1: No cohesive interaction is present

PRCI = 0/12 = 0
NRCI = 0/12 = 0
ORCI = 0/12 = 0

Case 2: All possible cohesive interactions are present

PRCI = 12/12 = 1
NRCI = 12/12 = 1
ORCI = 12/12 = 1

Case 3: Incomplete interaction graph

PRCI = 4/7 = .571
NRCI = 4/5 = .8
ORCI = 6/7 = .857

Case 4: Isolated object
O1 has been added to Case 3. This decreases cohesion because O1 has no known interactions with the rest of the module data declarations and subprograms.

PRCI = 4/12 = .333
NRCI = 4/7 = .571
ORCI = 9/12 = .75

Case 5: Single object

PRCI = 0/3 = 0
NRCI (= 0/0) = undefined
ORCI = 3/3 = 1

No information is available on the interactions between object O and the three subprograms. Therefore, ORCI and PRCI provide the bounds of the admissible range for cohesion, and NRCI is undefined, i.e., it could take any value in between. The more incomplete the information, the wider the uncertainty interval.

5 Module Coupling

According to commonly accepted design principles, design must show low coupling between modules. In this section, we first give general definitions and assumptions on coupling (Section 5.1). Then, we present a set of metrics (Section 5.2), and discuss the issue of genericity (Section 5.3) in the context of coupling.

5.1 Definitions

Definition 8: Import Coupling of a module (IC):
Import Coupling is the extent to which a module depends on imported external data declarations.

Assumption A-IC:
The more dependent a module on external data declarations, the more difficult it is to understand in isolation. In other words, the larger the amount of external data declarations, the more incomplete the local description of the module specification, the more spread the information necessary to isolate and understand a change. Thus, if there is a high average coupling within a set of modules, both activities *A1.1* and *A1.2* in Figure 1 are affected. The design of the change (phase *P2* in Figure 1) is also more complex.

Definition 9: Export Coupling of a module (EC).
Export coupling is the extent to which a module's internal data declarations affect the data declarations of the other modules in the system.

Assumption A-EC:
Export coupling is related to how a particular module is used in the system. As such, EC should have a direct impact on understanding the effect of a change on the rest of the system, and on validating the system after the change.
The larger the number of DD-interactions with external data declarations, the larger the likelihood of ripple effects when a change is implemented (activity *A4.2* in Figure 1). Also, the larger the number of potential DD-interactions, the more complex testing and verification become, since potential side effects have to be identified and addressed based on actual DD-interactions (activities *A4.1* and *A4.2* in Figure 1).

The import coupling of a module will be expressed in terms of the actual DD-interactions between imported/visible external data declarations (i.e. global) and the internal data declarations of the module. Export coupling will be based on both the actual and potential DD-interactions between locally defined data declarations and the other data declarations within the scope of the module. Actual DD-interactions are important because they capture the actual dependencies between a module and its context of declaration and therefore should be closely related to the likelihood of ripple effects. According to the defined assumption, the number of potential DD-interactions of a module with its context of declaration should be related to the ease of verifying and testing the side effects of the implemented change. These potential DD-interactions will simply be determined by the programming language visibility rules.

5.2 Metrics Based on Coupling

The issue will be first addressed by ignoring generic modules for the sake of simplification. Generic modules and their impact on the defined metrics will be treated in Section 5.3.

Definition 10: Global versus Locally defined data declarations
We will denote by *Global(m)* the set of all the external data declarations imported by a module *m*, and by *Local(m)* the set of all the locally defined data declarations in module *m*.

Definition 11: Scope of a module
Scope(m) is the set of all data declarations declared outside the module for which the internal data declarations of module *m* are visible.

Definition 12: Import Coupling
We will use the following metric to capture Import Coupling

IC(m) = DD-interactions(Global(m), Local(m))

In the above definition, we have considered all sort of imports equally. However, in terms of impact on the change difficulty in a particular module, imports from the same hierarchy or the same subsystem do not equate with imports from outside the module's hierarchy or subsystem. There are several reasons for this, e.g., people may have a better familiarity with the subsystem they are in charge of maintaining, understanding a module in another hierarchy increases the load of information to be known for understanding the change. Although we do not fully investigate this complex issue here, a simple solution to refine *IC* could be to define all the metrics presented below separately for several categories of coupling, e.g., coupling with modules outside the subsystem.
Each box in Figure 6 represents a module specification. Submodule specifications *C2* and *C3* are located in their parent's body *C1*. *C2* is assumed to be declared before *C3* and therefore visible to *C3*. The *Inst()*, *Sub()*, *Derived()*, *ValDep()* and *Const()* functions specify if one data declaration is respectively the object instantiation of a type, a subtype of a type, the derived type of another type, an object dependent on the value of another object (e.g. initialization), an object used to constrain a type or another object definition. Note that the same data declaration may interact with several data declarations, e.g., *T21* in Figure 6. *Tij* and *Oij* data declarations represent respectively types and objects in module *Ci*. *FPij* represents subprogram formal parameters. Even though they are objects, we identified them by a different symbol in order to improve the figure readability. The *IC* values for the modules in Figure 6 are computed as follows

IC(m) = direct DD-interactions + transitive DD-interactions

IC(C1) = 0 + 0 = 0
IC(C2) = 3 + 1 = 4
-- from C1 (direct: O11 twice,T12; transitive: T21)
IC(C3) = 2 + 2 = 4
-- from C1 (direct: T12; transitive: T12 twice) and C2 (direct: T21)
IC(C4) = 1 + 0 = 1 -- from C1 (direct: T11)

Definition 13: Potential and Actual Export Coupling
As presented in the assumption *A-EC*, both actual and potential coupling need to be measured.

EC-Actual(m) = DD-interactions(Local(m), Scope(m))

EC-Potential(m) = |Local(m)| · |Scope(m)|

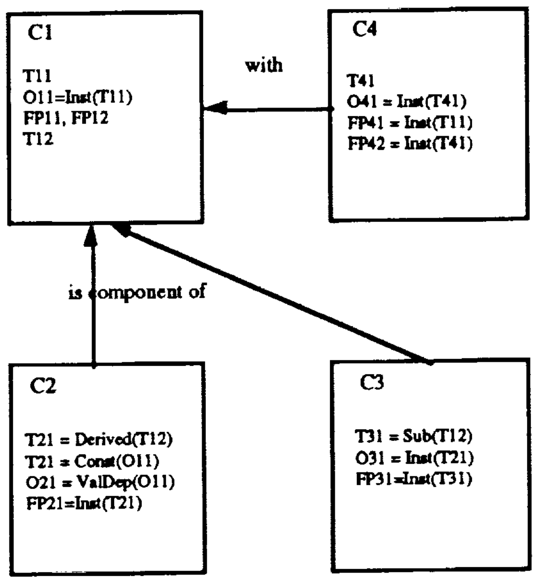

Figure 6: Calculation of IC and EC with non-generic components only

In the example of Figure 6, illustrated by the results presented below, we see that *C1* expectedly shows the largest actual and potential export coupling.

EC-Actual(m) = direct DD-interactions + transitive DD-interactions

EC-Actual(C1) = 5 + 3 = 8
-- to C2 (direct: T12, O11 twice; transitive: T12),
-- to C3 (direct: T12; transitive: T12 twice),
-- to C4 (direct: T11)
EC-Actual(C2) = 1 + 1 = 2
-- to C3 (direct: T21; transitive: T21)
EC-Actual(C3) = 0 + 0 = 0
EC-Actual(C4) = 0 + 0 = 0

EC-Potential(C1) = 5 · 10 = 50
EC-Potential(C2) = 3 · 3 = 9
EC-Potential(C3) = 3 · 0 = 0
EC-Potential(C4) = 3 · 0 = 0

We now introduce a normalized measure, Relative Dependency, to capture how dependent a module is on external data declarations with respect to the whole set of data declarations it can access, i.e., the external data declarations and its own data declarations. This normalized measure may contribute to capture the difficulty of the understanding process described in assumption *A-IC*, along with the absolute *IC* measure.

Definition 14: Relative Dependency (RD)
The relative dependency of a module *m* is the ratio of Import Coupling normalized by the total number of DD-interactions, i.e., within *m* itself and between the data declarations external to m and m.

RD(m) = IC(m)/(DD-interactions(Local(m), Local(m)) + IC(m))

RD(m) is therefore a unitless measure of import coupling of the module with the rest of the system which is relative to the total number of DD-interactions. Thus, a large module with a large import coupling might show a somewhat low relative dependency.

For Figure 6, we obtain the following results:

DD-interactions(C1, C1) = 1
DD-interactions(C2, C2) = 1
DD-interactions(C3, C3) = 1
DD-interactions(C4, C4) = 2

RD(C1) = 0/(1+0) = 0
RD(C2) = 4/(1+4) = 0.8
RD(C3) = 4/(1+4) = 0.8
RD(C4) = 1/(2+1) = 0.33

We can differentiate two main families of modules, based on IC and EC: "servers", i.e., provider of services, and "clients", i.e., users of services.

Definition 15: Coupling type (CT)
The coupling type of a module *m* is the ratio of Import Coupling normalized by the total Export and Import Coupling of module *m*.

CT(m) = IC(m)/(EC-Actual(m) + IC(m))

When *CT < 0.5*, then the module is more of the type "server"; otherwise, it can be classified as a "client". The first type of modules is expected to be more often at the top of the system hierarchies while the second type should be more common at the bottom of those hierarchies. This is what happens in the example in Figure 6, as the results presented below show.

CT(C1) = 0/(8+0) = 0	-- server
CT(C2) = 4/(2+4) = 0.66	-- more of the client type
CT(C3) = 4/(0+4) = 1	-- client
CT(C4) = 1/(0+1) = 1	-- client

Exceptions to this pattern may be the symptom of anomalies in system design.

5.3 The Treatment of Generic Modules

There are two possible ways of taking into account generics when calculating coupling. Either each instance may be seen as a different module or a generic may be seen as any other module whose scope/global data declarations is/are the union of the scope/global data declarations of its instances. The second solution does not consider instances as independent modules and appears to be more suitable to our specific perspective (i.e., the change process) since instances cannot be modified directly and only one module is to be maintained: the generic module. In other words, if *N* instances are generated, we will not count coupling as if *N* modules were actually developed since those instances may only undertake change through their corresponding generic module. Generic formal parameters allow for the substitution of objects, types and subprograms. This substitution does not have any impact on the number and the kind of exported data declarations (i.e. same number of type, object declarations respectively imported and exported).
When calculating import coupling, we will count the DD-interactions of the generic modules with the union of the global data declarations specific to their instances. When calculating export coupling, we will count the DD-interactions of the generic modules within the union of the scope of their respective instances. Consistent with the definition of DD-interaction, generic formal parameters DD-interact with their particular generic actual parameters (i.e. type, object) when the generic module is instantiated since a change in the former may imply a change in the latter.
This is what the following example illustrates. The graphical formalism is identical to the one used in Figure 6 and function *New(G, P)* represents a new instantiation of a generic package or subprogram *G* with a generic formal parameter *GFP1* and its generic actual parameter set *{P1, P2}*.
C2 and *C3* only import data declarations from *G* (with *TG1*). *C1* imports from *G* (*P1*, *P2* DD-interact with *FGP1*).

IC(m) = direct DD-interactions + transitive DD-interactions

IC(C1) = 2 + 0 = 2	-- from G
IC(C2) = 2 + 1 = 3	-- from G and C1
IC(C3) = 3 + 1 = 4	-- from G and C1
IC(G) = 0 + 0 = 0	

EC(m) = direct DD-interactions + transitive DD-interactions

EC-Actual(C1) = 2 + 2 = 4 -- to C2, C3
EC-Actual(C2) = 0 + 0 = 0
EC-Actual(C3) = 0 + 0 = 0
EC-Actual(G) = 5 + 0 = 5 -- to C1, C2, C3

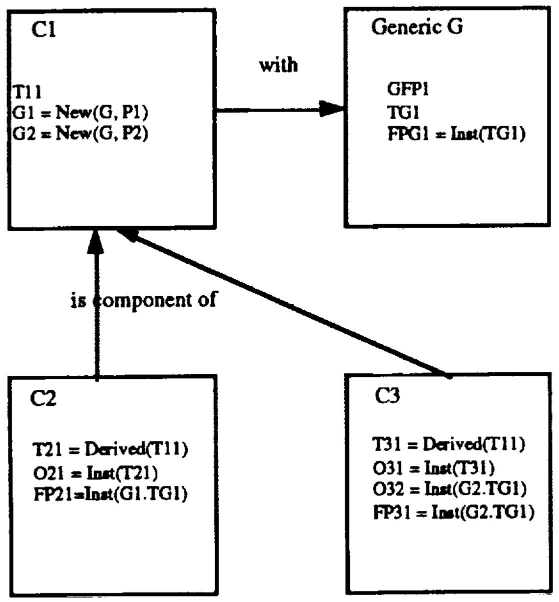

Figure 7: Generics when calculating coupling

The RD metric shows that G is the only fully independent module. The others strongly depend on external data declarations:

DD-interactions(C1) = 0
DD-interactions(C2) = 1
DD-interactions(C3) = 1
DD-interactions(G) = 1

RD(C1) = 2 / (0 + 2) = 1
RD(C2) = 3 / (1 + 3) = 0.75
RD(C3) = 4 / (1 + 4) = 0.8
RD(G) = 0 / (1 + 0) = 0

6 A Visibility Control Metric Based on Coupling

As opposed to the metrics presented in previous sections, this metric does not characterize modules but sets of modules. Here, we want to assess to which extent visibility is controlled in the design of a system, subsystems or any system part [G86, AE92]. Thus, we want to identify design flaws related to visibility.

Assumption A-VC:

If the system, the subsystem or the hierarchy has been designed by following minimal visibility rules, modules with larger potential export coupling should also have larger actual export coupling. This is the case in the above example where the ranking according to *EC-Potential* is identical to the ranking by *EC-Actual*. Therefore, we want to measure the correlation between *EC-Actual* and *EC-Potential* in order to determine whether or not highly visible modules are also highly used modules. In other words, this can be interpreted as how well visibility is controlled within the system or a part thereof.

Remark.
We do not intend to judge the designer work through this process, since other constraints may bias the design towards a non optimal visibility control. We look at it from the narrow perspective of the change process, leaving to the designer the decision of possible tradeoffs between maintainability and other criteria, e.g., performance.

We do not want the measure of correlation to be based on parametric assumptions since we do not know what kind of relationship to expect between actual and potential export coupling[CAP88]. One way of doing it is to use a non-parametric statistic which takes into account the rank of each module with respect to both *EC-potential* and *EC-actual*. This type of statistic does not require any functional assumption and is moreover robust to outliers. Thus, we will be protected against illusory strong correlations due to outliers and falsely weak correlations due to wrong functional assumptions. If visibility is close to minimal, we assume the ranks of the modules to be similar with respect to those two metrics.

Definition 16: Visibility control (VC)
The visibility control of a set of modules *SM* (*VC(SM)*) is measured by means of the Spearman's rank correlation coefficient [CAP88] between the actual Export Coupling and the potential Export Coupling

$$VC(SM)=1-[\Sigma_{m\in SM}(D^2(m))/(|SM|(|SM|^2-1)/6)]$$

where D(m) = Rank(EC - Actual(m)) - Rank(EC-Potential(m))

The larger *VC(SM)*, the closer to minimal the visibility. When there is no association, it can be shown that $\Sigma_{m \in SM}(Distance^2(m)) = (|SM|(|SM|^2 - 1) / 6)$, so $VC(SM) = 0$.

7 Conclusions

In this paper, we have presented a comprehensive approach for evaluating the high-level design of software systems which is summarized by the following characteristics:

- early available metrics based on precisely defined assumptions and related without ambiguity to the defined change process model
- definitions of module cohesion, module coupling and visibility control consistently based on the notion of interaction, which is closely related to the phenomenon of change side effects
- an OOD [BO87] view of a software module as opposed to the usual subroutine perspective [M77, YC79] of coupling and cohesion evaluation
- a clear separation between Ada-specific and language-independent concepts.

Our future research will encompass:

- the definition and refinement of other higher-level metrics based on module coupling and cohesion that will characterize higher-level constructs, e.g., module hierarchies, subsystems.
- the experimental validation of the proposed metrics with respect to change difficulty (i.e., man-hours) and size (i.e., number of modules changed, lines of code removed, changed, added).
- the development of high level metrics based on other software engineering principles, such as information hiding and reuse.

Acknowledgments

We thank Bill Thomas and Chris Hetmanski for their helpful comments on the earlier drafts of this paper.

References

[AE92] W. Agresti and W. Evanco, "Projecting Software Defects from Analyzing Ada Designs", *IEEE Trans. Software Eng.*, 18 (11), November, 1992.

[BB92] L. Briand, V. Basili, "A Classification Procedure for the Effective Management of Changes During the Maintenance Process", Conference on Software Maintenance, 1992, Orlando, Florida.

[BO87] G. Booch, "Software Engineering with Ada", Benjamin/Cumming Publishing Company, Inc., Menlo Park, California, 1987.

[BR88] V. Basili and H. Rombach,"The TAME Project: Towards Improvement-Oriented Software Environments", *IEEE Trans. Software Eng.*, 14 (6), June, 1988.

[CAP88] J. Capon, "Elementary Statistics", "Statistics for the Social Sciences", Wadsworth Publishing Company, 1988.

[CY79] E. Yourdon, L. Constantine, "Structured Design", Prentice Hall, 1979

[DoD83] ANSI/MIL-STD-1815A-1983, Reference Manual of the Ada Programming Languages, U.S. Department of Defense, 1983

[G86] J. Gannon, E. Katz, V. Basili, "Metrics for Ada Packages: an Initial Study", Communications of the ACM, Vol. 29, N. 7, July 1986.

[G92] C. Ghezzi, M. Jazayeri, D. Mandrioli, "Fundamentals of Software Engineering", Prentice Hall, Englewood Cliffs, NJ, 1992

[HK84] S. Henry, D. Kafura, "The Evaluation of Systems' Structure Using Quantitative Metrics", Software Practice and Experience, 14 (6), June, 1984.

[IS88] D. Ince, M. Shepperd, "System Design Metrics: a Review and Perspective", Proc. Software Engineering 88, pages 23-27, 1988

[K88] B. Kitchenham, "An Evaluation of Software Structure Metrics", Proc. COMPSAC 88, 1988

[M77] J. Myers, "An Extension to the Cyclomatic Measure of Program Complexity", SIGPLAN Notices, 12(10):61-64, 1977

[R87] H. D. Rombach, "A Controlled Experiment on the Impact of Software Structure and Maintainability:", *IEEE Trans. Software Eng.*, 13 (5), May, 1987.

[R90] H. D. Rombach, "Design Measurement: Some Lessons Learned", *IEEE Software*, March 1990.

[S90] M. Shepperd, "Design Metrics: An Empirical Analysis", *Software Engineering Journal*, January 1990.

[SB91] R. Selby and V. Basili, "Analyzing Error-Prone System Structure", *IEEE Trans. Software Eng.*, 17 (2), February, 1991.

[Z91] W. Zage, D. Zage, P. McDaniel, I. Khan, "Evaluating Design Metrics on Large-Scale Software", SERC-TR-106-P, September 1991.