Testability Analysis of a UML Class Diagram[1]

Benoit Baudry, Yves Le Traon, and Gerson Sunyé
IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France
{Benoit.Baudry, Yves.Le_Traon, Gerson.Sunye} @irisa.fr

***Abstract**.*
Design-for-testability is a very important issue in software engineering. It becomes crucial in the case of OO designs where control flows are generally not hierarchical, but are diffuse and distributed over the whole architecture. In this paper, we concentrate on detecting, pinpointing and suppressing potential testability weaknesses of a UML class diagram. The attribute significant from design testability is called "class interaction": it appears when potentially concurrent client/supplier relationships between classes exist in the system. These interactions point out parts of the design that need to be improved, driving structural modifications or constraints specifications, to reduce the final testing effort.

1 Introduction

Software testing is often a very costly part of its life cycle. Any technique that improves a software design at an early stage can have highly beneficial impact on the final testing cost and efficiency. This paper is concerned with the issue of testability of object-oriented (OO) static designs based on the UML (Unified Modeling Language) class diagrams. It aims at pinpointing the parts of the software architecture where problems due to undesired interactions must be tested. This question of testability [1] has been revived with the object-orientation [2,3]. Object-orientation is now widely chosen by software industry, despite that the technology is not mature enough from the testing point of view.

To guide the testing task, the main OO static design view, namely the class diagram, appears as a good basis to detect and master the widespread implicit control dependencies, due to inheritance and dynamic binding. However, a class diagram is often ambiguous, incomplete, and may lead to several false interpretations, consequently possibly false implementations and, dramatically, useless tests. Complementary views of the UML, such as object diagrams or collaboration diagrams and sequence ones could help. Indeed, collaboration diagrams may serve as expected traces that a test case must exhibit [4], while sequence diagrams offer a basis for specifying nominal and exceptional test purposes. If statechart diagrams represent exhaustively a given dynamic behavior, collaboration and sequence diagrams may help understanding interactions but cannot detail each one nor restrict their possible number. In the same way, object diagrams only represent a particular system configuration of class instances and do not catch all potential ones. In conclusion, we consider that the main views on which testability must be analyzed are class diagrams and statecharts, while the other views only display snapshots of some possible behaviors. This work focuses on the testability weaknesses of UML class diagrams.

Testability problems in a class diagram are due to the existence of client/supplier relationships in the system, like for any classical software. Indeed, if there were no client in the software there would be no defined set of executions and thus nothing to test. Thus, after unit testing, failures should only occur because of a misuse due to wrong interactions between objects: these interactions go throughout the architecture and are made more complex if the client/supplier dependencies traverse inheritance trees. Polymorphic dependencies multiply the number of potential object types that may interact with various - and possibly false - implementations. This paper introduces a testing criterion that requires the coverage of these object interactions. To be realistically applied, the number of test cases must be reasonable and the paper proposes an estimate of the testing effort, measured by approximating the number of object interactions from the UML class diagram. The estimate we consider as significant from the overall testability of a class diagram is the number of "class interaction": a class interaction is a topological configuration in the class diagram on which testing has to focus as a hard point. It occurs if a class is supplier from another through various possible paths of dependencies.

Based on the proposed testing criterion, the objectives of the paper are to:

- provide a model to capture class interactions and pinpoint classes that cause the interactions,
- measure the number and complexity of the interactions due to polymorphic uses, considered here as our estimate of design testability,
- suggest improvements on the design to reduce the number and complexity of class interactions: these

[1] This work has been partially supported by the CAFE European project. Eureka S! 2023 Programme, ITEA project ip 0004.

0-7695-1339-5/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

improvements at design level are realistic since static verifications on the code ensure their implementation,
- *in fine* provide a way of accepting or rejecting a design based on testability analysis. The design is rejected when no improvement can be added to limit the object interactions.

Section 2 opens with an analysis of testing problems due to potential misuses in the implementation of an object-oriented architecture. It also describes a correlated testing criterion that brings a basis for analyzing design testability: a design-for-testability methodology is proposed. Section 3 gives the graph model derived from a class diagram, which captures potential interactions. Section 4 details a reduced set of refinement actions to master and reduce the number of these interactions: 4 UML stereotypes are defined for that purpose. Section 5 illustrates the model on typical architectures.

2 Testing interactions in an object-oriented system

In this section, we suggest a testing adequacy criterion for object-oriented systems that aims at detecting object misuses due to erroneous interactions. Based on the UML, as a reference specification, and being given an implementation under test, it aims at covering all object-to-object dependencies that should be tested. For sake of clarity, the class diagram is the main specification used to define precisely what must be tested. To apply the criterion, we show that the design must be precise enough and as close as possible to the actual implementation. Testability problems are emphasized on a class diagram: design with an unreachable testing goal can be either improved or rejected as not testable. This testability analysis methodology is presented and the following sections will detail each of its steps.

2.1 Testing adequacy criterion for OO systems

In an object-oriented system, if an object o_1 manipulates another object o_2, by a direct or transitive path of dependencies, test cases should cover all usage of o_2 by o_1. However, the number of such dependencies is prohibitive for most systems and leads to unrealistic testing criteria [5]. Here, we concentrate on hard-to-detect errors that appear when non-expected side effects occur, i.e. when one or several objects may modify the state of an object using independent paths of dependencies. In the following, we provide a non-ambiguous definition to the intuitive notions of *dependency* and *path* (cf. section 3.1). The testing criterion we propose concerns combination of dependencies that could lead to inconsistent state for either of the objects that depend on each other. Let us describe the testing criterion based on the UML class diagram. We introduce the concept of "class interaction" when potentially concurrent client/supplier relationships between the same classes along different paths exist in a system

Classes depend on each other's for their processing. A class A is said to use another class B if it calls methods from B, either through an attribute of type B or a local variable. The UML allows illustrating this relationship on a class diagram either by drawing an association between the classes or a dependency with the stereotype «uses». This relationship is called a direct usage relationship between classes.

Direct usage relationship. *There is a direct usage relationship from class A to class B on a UML class diagram, if there exists an association or a «uses» dependency from A to B. In case of non-directed associations, dependencies exist from A to B and from B to A. The set of direct usage relationships for a class diagram is denoted SDU.*

Section 3.1 provides a set of rules to derive dependencies from UML main class diagram features. Now, the direct usage relationship is extended to transitive usage relationship. Yet, a relationship may exist between two classes A and B even if there is neither an association nor a dependency between them due to transitive relationships.

Transitive usage relationship. *The transitive closure of SDU defines all transitive usage relationships between classes of the class diagram. The ith transitive usage relationship from class A to class B is denoted A $\mathcal{R}_i$ B. If the final code allows the instantiation of a transitive usage relationship from an object o_1 of class A to an object o_2 of class B, we say there is a real transitive relationship from A to B.*

We now define the notions of class interaction and object interaction. The first one is a potential interaction since it is detected from the class diagram which is only an abstract view of the software. Indeed, the interactions detected at the design level can disappear or can worsen when the design evolves and is implemented. The second one is a real interaction since relationships between running objects are involved. Some of them can be detected at the design level from Object diagrams, or sequence diagrams, but, since those diagrams can offer only a partial view of the system, and are likely to change, they cannot be used to detect every real interactions in the system

Class Interaction (potential interaction). *An interaction from class A to class B occurs iff:*

$\exists$ *i and j, $i \neq j$, such as A $\mathcal{R}_i$ B and A $\mathcal{R}_j$ B.*

It has to be noticed that a class interaction may involve more than two transitive usage relationships.

Object Interaction (real interaction). *There exists an object interaction from an object o_1 of class A to o_2 of class B iff:*
- *there exists i and j, $i \neq j$, such as A $\mathcal{R}_i$ B and A $\mathcal{R}_j$ B,*

IEEE COMPUTER SOCIETY

- R_i *and* R_j *are real transitive relationships for* o_1 *and* o_2.

Property. *The number of class interactions is an upper bound for the number of object interactions.*

The property is obvious under the assumption that the code is derived (possibly automatically using an appropriate CASE tool) from the design.

Testing criterion. *For each class interaction, either a test case is produced that exhibits a corresponding object interaction, or a report is produced that shows this interaction is not feasible.*

The task of producing test cases/reports is impossible if the number of class interactions is high. The main purpose of the paper concerns the limitation of these interactions by improving the design. Indeed, the design must be as close as possible to the code. Hopefully, we have not to deal with the determination of real interactions: even with code, the real dependencies cannot be statically deduced, since OO languages are not strongly typed. Since the number of class interactions is an upper bound of the number of object interactions, we recommend putting additional information on the design that would reduce the number of class interactions. These additional information are design constraints for the programmer (e.g. expressed using UML stereotypes): one can statically verify that the implementation meets the constraints. This means that using static verification at the code level reduces the testing effort. As an example, being given a «create» stereotype on a dependency from A to B, the code of class A should invoke only the creation methods of B. This can be verified statically.

2.2 Informal Analysis of Testability Weaknesses

In order to informally study the problem due to class interactions that can appear when testing an OO system, we study a real object-oriented architecture, which class diagram is given in Figure 2.

This software allows distant instant messaging clients to communicate using the ICQ protocol. Any kind of media may be used: texts, sounds, and video. There are two central classes in this architecture, CLIENT and BUDDY. Both classes can be either in a connected or non-connected state. An instance of CLIENT is connected to a BUDDY via a direct or indirect protocol, depending on the state of the `buddy`.

This architecture is a typical object-oriented design. It uses basic constructs of object-orientation: inheritance, interfaces, abstract classes, and usage dependency relationships between classes in the system. A first look at this architecture reveals that many classes have strongly inter-dependent processes. For instance, all the children classes are strongly linked to their parent classes, and CLIENT and CLIENTSTATE are interdependent. This type of architecture has a considerable potential for faulty behavior. For example, BUDDY may depend on AIMDIRECTPROTOCOL via several paths. If such usage is undesired, it has to be either tested for, or avoided by constrained construction. These potential problems have to be recognized in order to estimate the verification and validation effort. The two potential sources of problems are the following:

- When a method `m1` in class CLIENT uses a method `m` of class CONNECTED, the class CONNECTED may use CLIENT to process `m`. That means that the class CLIENT might use itself when it uses CONNECTED to process part of its work.
- When a class of BUDDY uses ICQDIRECTPROTOCOL, it might do so in two different ways: directly by declaring an instance of class ICQDIRECTPROTOCOL, or through a use of CONNECTED which uses DIRECTPROTOCOL which can then be instantiated by ICQDIRECTPROTOCOL.

The exact number of potential misuses as well as their complexity is difficult to determine with a simple observation of the design. Thus, we need a model to capture all these interactions with the inheritance complexity.

This informal analysis emphasizes two potential testability weaknesses: interactions from one class to another, and a configuration we call *self-usage* that corresponds to a class that uses itself by transitive usage dependencies. Both problems worsen when usage dependencies go through an inheritance tree because of polymorphism. Next section illustrates this point.

2.3 Inheritance complexity

The complexity due to inheritance appears when transitive dependencies go through one or several inheritance hierarchies. On Figure 1, there is a class interaction from C to D. The interaction is complex because if C uses an instance of class a or A2 or A21, anyway those three classes have relationships between each other. In that case, the interaction with each of the three potential usages by C (A or A2 or A21) have to be tested, and for each of those, we have to test the relationships between the classes in the inheritance hierarchy. However, by constraining the design (and make it more precise), we can reduce the complexity of the interaction. Indeed, if classes A and A2 are interface classes, we can ensure that the C can only use A21 or A22: the area of the interaction with class D is thus reduced to class A21. The model must also capture the complexity of the interaction.

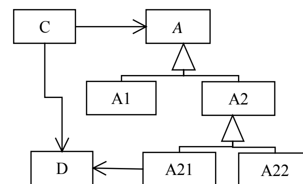

Figure 1 - Concurrent usage through an inheritance hierarchy

IEEE COMPUTER SOCIETY

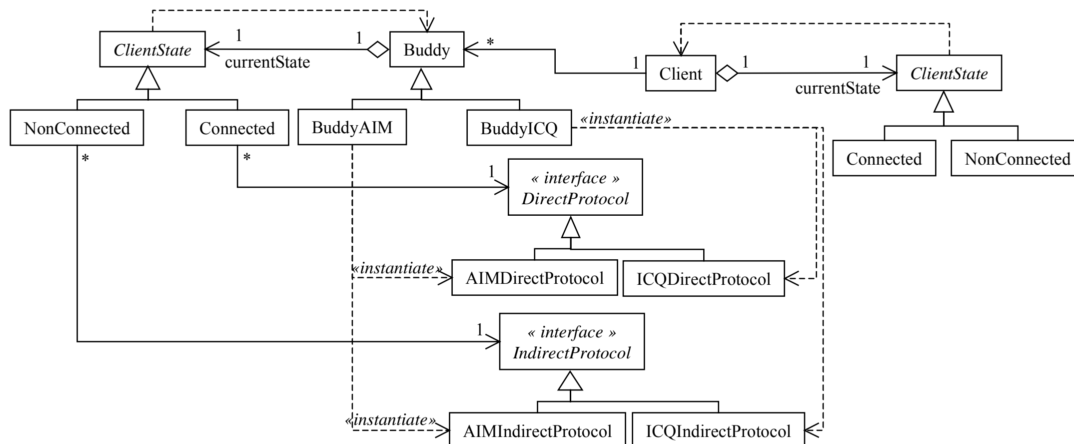

Figure 2 - An Instant-Messaging Client

The testing model has thus to discriminate between up and down dependencies into an inheritance tree. Moreover, the testing model must not count brother classes as dependent, since they are always independent from a testing point of view.

2.4 Designing for testability: a Methodology

Figure 3 summarizes an approach that helps improve design's testability, based on the testing criterion proposed in section 2.1. An initial object-oriented design (the global UML class diagram of the software) is modeled to compute the number of class interactions in the system as well as their complexity (1). Once this number is available, one can decide it is too high:

- the designer can improve the design (2),
- the design may be rejected as untestable with respect to the test criterion (3).

The design improvement can be done, either by reducing coupling in the architecture [6], or by expressing constraints that will help the developer avoid implementing error-prone object interactions. Our suggestion is to use dedicated stereotypes on association and dependencies specifying more clearly the type of usage that must be implemented (creation or reading …).

When the design meets the testability requirements, it can be implemented (4) and the constraints added for testability improvement of the design must be verified before testing (5).

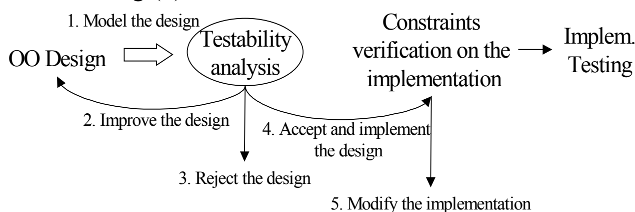

Figure 3 - Improving testability of OO designs

Section 3 details the model for testability analysis (1), section 4 presents possible improvements to reduce the complexity of interactions, and thus improve testability (2).

3 Modeling class interactions

In previous section we have expressed the need for an abstract model of a class diagram on which it would be easier to pinpoint all the class interactions for a system. The model we propose is based on a graph representation of the system. In this section, we give rules to build a graph from a UML class diagram. Such a graph is called Class Dependency Graph (CDG). Definitions are given to define this graph, Then, topological rules on the graph are given which formally determine potential interactions. The CDG serves as a basis to apply classical graph algorithms to detect interactions and measure their complexity.

3.1 Graph construction from a UML model

In this section, several definitions about the class dependency graph model are given, especially about how to build it, and what information it carries.

In the following definitions, we call $\mathcal{C}$ the set of all the classes of a system, and M(C) the set of the methods of a class $C \in \mathcal{C}$.

Class Dependency Graph (CDG). *A class dependency graph is a pair CDG = (X,Γ), where*

- X is the set of vertices*, each vertex representing a class of an object-oriented system. A class is represented by a single vertex.*

- Γ is the set of pairs $(x,y) \in X^2$, called set of directed edges ($(x,y) \neq (y,x)$). An edge between two vertices, x and y, represents a dependency between from class x to class y. An edge is labeled by the type of dependency that exists between the two classes, namely usage dependencies or inheritance.

A CDG can be easily built from a UML class diagram, the transformation rules are given Figure 4. These transformations are made explicit in the following definitions.

Proceedings of the Eighth IEEE Symposium on Software Metrics (METRICS'02)
0-7695-1339-5/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

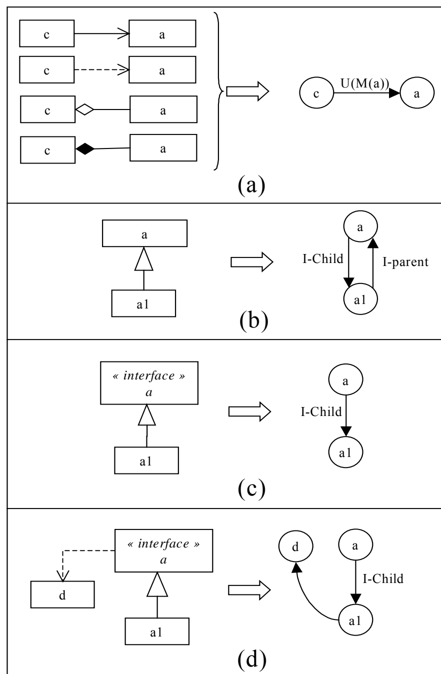

Figure 4 - Basic transformations from a UML class diagram to a CDG

Note that, since there is a vertex for each class and each vertex represents one and only one class, in the following definitions, the vertex corresponding to a class c is simply called c.

Edge labels. *Every edge in a CDG represents a dependency between two classes of an object-oriented system. The edge between vertices* $C \in \mathcal{C}$ *and* $D \in \mathcal{C}$ *is labeled by the type of dependency that exists between* C *and* D*. Dependencies can be of two types: usage (label U) if* C *uses* D*, or inheritance (label I) if* $C \neq D$ *and* C *inherits from* D*.*

Label U. *We associate a set of methods to the label U which corresponds to the set of methods in M(d) used by class* C*. The default value of this set of methods is M(*D*) (as long as we do not know the sub-set of M(*D*) used by* C*). This transformation is illustrated Figure 4 (a).*

In the case of Usage dependency stereotyped «instantiate» or «create» between classes C and D, the set of methods associated to the label U is (`createD()`) and indicates that C only calls the creation method of class D through this usage relationship.

Label I. *The inheritance label is derived in two labels: I-child and I-parent (Figure 4(b)). If* $C \in \mathcal{C}$, $D \in \mathcal{C} - \{C\}$*, and* C *directly inherits from* D*, then there is an edge (d,c) labeled I-child and an edge (c,d) labeled I-parent.*

If D is a pure interface, the (c,d) edge does not exist (in that case, C can not use methods from D, Figure 4(c)), and if class D depends on other classes, then, the edges labelled U from D do not exist, but are moved to concrete children of D (Figure 4(d)).

From a testing point of view, we need a dependency from the parent to the child, because everywhere the parent class occurs, the child can occur as well. So, for every parent of the class, we must test the same statement with an occurrence of every child. The dependency from the child to the parent is obvious: c uses d when it calls a method m inherited from d.

Example: Figure 5 shows a class dependency graph obtained from a small class diagram, by applying transformation rules given in the definitions above.

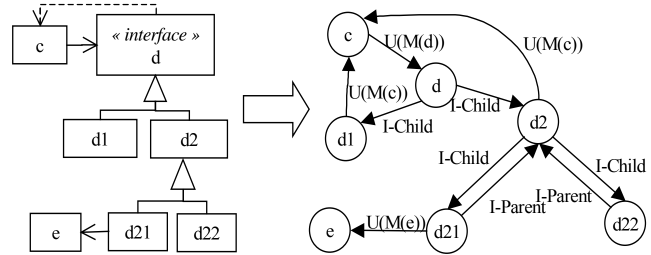

Figure 5 - CDG example

3.2 Detecting Class Interactions from the CDG

In this section, we first recall the classical definitions of path and cycles. We then come back to the potential interactions described in section 2.2, and precise their definitions in the CDG context. We also propose a computation of interactions' complexity based on the CDG.

Path. *A path P in a CDG is a sequence of vertices* $P=[x_{i_1}, x_{i_2}, x_{i_3},\ldots, x_{i_k}]$*, such that:*

- $(x_{i_1}, x_{i_2}) \in \Gamma, (x_{i_2}, x_{i_3}) \in \Gamma,\ldots, (x_{i_{k-1}}, x_{i_k}) \in \Gamma$
- x_{i_1} *is the origin of the path and is called origin(P)*
- x_{i_k} *the end and is called end(P)*
- *the* x_{i_j} $(2 \leq j \leq k-1)$ *are the intermediate vertices (the set of intermediate vertices is called itVertices(P)).*

Cycle. *Let P be a path, P is a cycle if and only if end(P)=origin(P).*

Elementary path, cycle. *An* elementary path *is a sequence of vertices in which there is never twice the same vertex. An* elementary cycle*, is an elementary path for which only the origin vertex is repeated.*

On the example of Figure 5, [c, d, d2, d22] or [c, d, d2, d21, e] are elementary paths, but [c, d, d1, d] is not. In the same way, [d, d1, d] is an elementary cycle, but [c, d, d1, d, c] is not.

3.2.1 Potential interactions.

Class Interaction (CI). *There exists a class interaction CI(c,d) from class c* $\in C$ *to class d* $\in C$-

IEEE COMPUTER SOCIETY

{c}, if there exists at least two elementary paths P_1 and P_2 such that $P_1 \neq P_2$ and $(origin(P_1) = origin(P_2)=c) \wedge (end(P_1) = end(P_2) = d) \wedge (itVertices(P_1) \neq itVertices(P_2))$.
A path going through an inheritance hierarchy must cross the hierarchy only in one direction, i.e. there must only edges going from child vertices to parent vertices, or only edges going from parent vertices to child vertices.

For example, in Figure 6, a potential CI(d,f) interaction can be detected because there are two different elementary paths going from d to f: [d,e,f] and [d,f] which intermediate vertices are distinct.

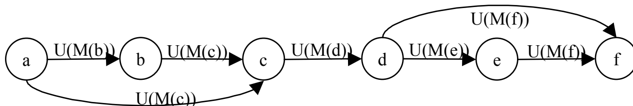

Figure 6 - CI on a CDG

This definition of the CI interaction takes into account only unitary interactions. For example, on the CDG of Figure 6, only two potential interactions are detected: CI(a,c) and CI(d,f), whereas the bigger interaction CI(a,f) is not detected. We assume that detecting only unitary interactions is sufficient, because solving CI(a,c) and CI(d,f) also solves CI(a,f).

Self Usage (SU). *There exists a self usage S (c) on class $c \in C$, if there exists an elementary cycle which origin is c.*
A cycle going through an inheritance hierarchy must cross the hierarchy only in one direction, i.e. there must be only edges going from child vertices to parent vertices, or only edges going from parent vertices to child vertices

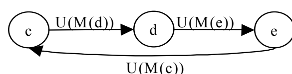

Figure 7 - SU on a CDG

Figure 7 shows a small graph on which a SU (c) interaction can be detected: there is an elementary cycle from vertex c to vertex c. As for the CI interaction, the definition of the SU interaction given above considers only unitary interactions.

3.2.2 Interactions complexity

The complexity of an interaction can now be formalized by taking into account polymorphism in the system. The interaction complexity increases when one or several paths involved goes through a graph corresponding to an inheritance hierarchy.

Complexity of interaction. *Let $P_1,\ldots, P_{nbPaths}$ be different paths corresponding to a class interaction CI. The complexity of the interaction is linked to the complexity of the different paths:*

$$complexity(CI) = \sum_{i=1}^{nbPaths} (complexity(P_i) \bullet \sum_{j>i} complexity(P_j))$$

The complexity of a path is defined in the following.

Descendents-path. *In an inheritance hierarchy, a descendents-path is the set of classes crossed by a path going from the root class to a leaf class.*

As defined earlier, paths involved in an interaction can go through an inheritance hierarchy only in one direction. Then we identify a sub-component corresponding to a slice of the inheritance hierarchy going from a root class to a leaf as shown Figure 8. This sub-component is called a descendents-path in an inheritance hierarchy. If a path involved in an interaction goes through one or several classes of a sub-component in the graph, the interaction's complexity grows in the following way: if there are n classes in the descendents-path, which are not pure interfaces, the complexity of the sub-component is n•(n-1). Every class has a relationship with each of the (n-1) others, and n•(n-1) interactions may occur that must be tested.

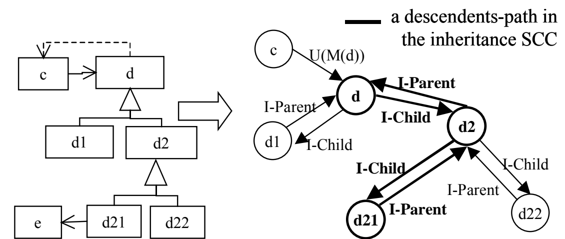

Figure 8 - Slice in a SCC corresponding to an inheritance hierarchy

The total complexity of a path is the product of the complexity associated to every hierarchy crossed by the interaction. Indeed, if two inheritance hierarchies are crossed, every class of one hierarchy can have a relationship with every class of the other hierarchy.

Complexity of a path in a class interaction. *Let P be a path involved in a class interaction, $IH_i,\ldots, IH_{nbCrossed}$ be the inheritance hierarchies crossed by P, the complexity of P is given as followed:*

$$complexity(P) = \prod_{i=1}^{nbCrossed} complexity(IH_i, P)$$

Next, let us define the complexity of a path going through an inheritance hierarchy.Several descendents-path in one inheritance hierarchy may increase the complexity of one path. If a path in the interaction goes through a class that is not a leaf in the inheritance hierarchy, there may be different descendents-path including this class. For example, on Figure 9, if an interaction goes only through class d2 in the inheritance hierarchy, the descendents-paths [d,d2,d21] and [d,d2,d22] are involved in the interaction.

Complexity of a path going through an inheritance hierarchy. *Let IH be an inheritance hierarchy and P be a path crossing IH. The complexity of IH for P is the addition of the complexity of $dp_1,\ldots, dp_{nbDP}$, the descendents-path in IH influencing P's complexity.*

$$complexity(IH,P) = \sum_{i=1}^{nbDP} complexity(dp_i)$$

IEEE COMPUTER SOCIETY

The complexity of a descendents-path corresponds to the number of potential interactions between classes in this path. In the worst case, each class in the class has a relationship with each other, so, if there are n classes in the path, there are at most n•(n-1) interactions in the path.

Complexity of a descendents-path. *Let dp be a descendent-path and h be the height of dp, the complexity for dp is:*

complexity(dp) = h • (h - 1)

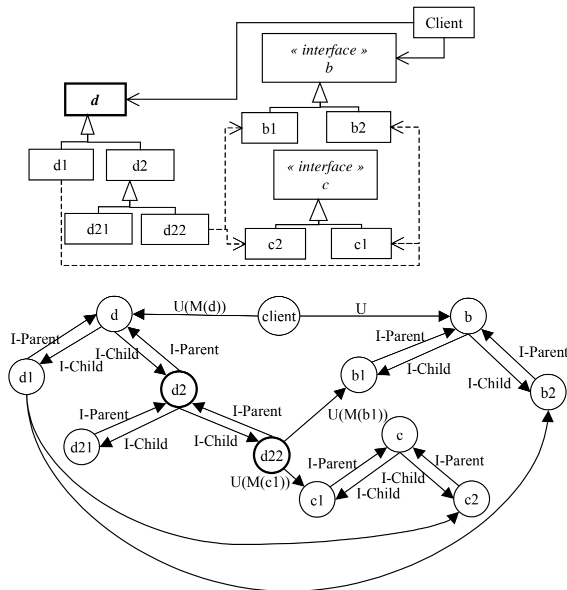

Figure 9 - a potential class interaction going through an inheritance hierarchy

For example, on Figure 9, there is a path going from the client class to the b1 class through the d inheritance hierarchy. The complexity of this path is 8: 1+3*(3-1)+1; 1 for the uses relationship from client to d, 3*(3-1) for the descendents-path of the d inheritance hierarchy, 1 for the uses relationship from d22 to b1.

4 Improving the Design Testability

Improving testability of the software, with respect to our testing criterion, means avoiding object interactions and especially concurrent accesses to shared objects. As we suggested in section 2.1, a solution may consist in clarifying the design, so that the code can be as close as possible to what the designer wants.

When it is possible, a way to improve testability, and break inheritance complexity, is the use of interface classes that are "empty" from an execution point of view: it is not possible in all cases. Besides, the UML allows a user to define *stereotypes* to associate a semantic to UML elements. We thus define several stereotypes that specify the semantic of links involved in class interactions (association, dependency, aggregation, composition). Thanks to these additional specifications, the programmer should avoid implementing an object interaction. As it will be illustrated in Section 5, this simple set of refinement actions may be of great help to improve the design, suppress ambiguity and reduce the testing effort. The stereotypes introduced here are analogous in some way to data flow testing criteria for classical software [7], that identify "definition" and "use" of variables in a program. This classical testing model aims at determining the data flow, the "life line" of variables at unit level. In an OO system, the designer gives the information: it aims here at determining the "communication lines" of objects throughout the system.

Here are the four stereotypes we propose:

- «create»: a create stereotype on a link from A to B means that objects of type A calls the creation method on objects of type B. If no use stereotype is attached to the same link, only the creation method can be called.
- «use»: a use stereotype on a link from A to B means that objects of type A can call any method excluding the create one on objects of type B. It may be refined in the following stereotypes:
  - «use_consult»: is a specialization of «use» stereotype where the called methods do never modify attributes of the objects of type B.
  - «use_def»: is a specialization of «use» stereotype where at least one of the called methods may modify attributes of the objects of type B.

Remark: By default, no stereotype on a link is equivalent to a combination of «use» and «create».

The stereotypes are taken into account by the graph model by associating another value to U labels. This also allows estimating the improvement of the design after adding stereotypes. It corresponds to step 2 of the methodology proposed in section 2.4.

To map these notions in the formal model, we take into account stereotypes in the interaction computation as follows: A class interaction exists if there exist two paths P1 and P2 of the interaction, such as:

- e1 being the entry edge of end(P1), e2 being the entry edge of end(P2),
- e1 and e2 have associated stereotypes «use» or «use_def»

For the other paths, their complexity does not participate to the complexity of the interaction. In that case, a clear separation is done in the use of shared provider. Each path of the interaction can be tested independently using Alexander and Offutt's testing criteria [5] (see related work).

Automated verifications may check that the code is in conformance with stereotypes constraints. For example, the verification of a «use-consult» from A to Bconsists in verifying that:

- A only calls query methods of B,
- B query methods never modify B state (directly and indirectly through the call of non-query methods).

IEEE COMPUTER SOCIETY

Next section illustrates potential testability problems on two small architectures, and gives examples of what can be done to avoid real problems at the code level.

5 Application examples

In this section, we apply the proposed testability analysis on two different designs, and for each of them, we propose advises that could improve the testability of these designs. First, we illustrate our approach with a micro architecture, an application of the Abstract Factory design pattern [8], then we study a typical compiler architecture. The obtained results are useful since they underline the hard points of the designs, where misleading interpretations may occur leading to a very hard to test implementation.

5.1 Testability Abstract Factory Design Pattern

Figure 10 shows an application of the Abstract Factory design pattern [8]. The CDG obtained from this class diagram is given Figure 11 (a). From this graph, it is easy to detect a potential class interaction from the CLIENT class to each of the WINDOW and SCROLLBAR classes: the CLIENT class can use them directly or through the “WIDGETFACTORY” inheritance hierarchy.

General data

$h_{AbsFact}$: height, in the WIDGETFACTORY inheritance hierarchy, of a descendents-path crossed by the CI.

h_{Prod}: height, in the WINDOW inheritance hierarchy, of a descendents-path crossed by the CI.

Complexity of the CI(CLIENT,WINDOW) in the abstract factory pattern:

$$(h_{AbsFact} \bullet (h_{AbsFact} - 1) \bullet h_{Prod} (h_{Prod} - 1)) \bullet (h_{Prod} \bullet (h_{Prod} - 1))$$

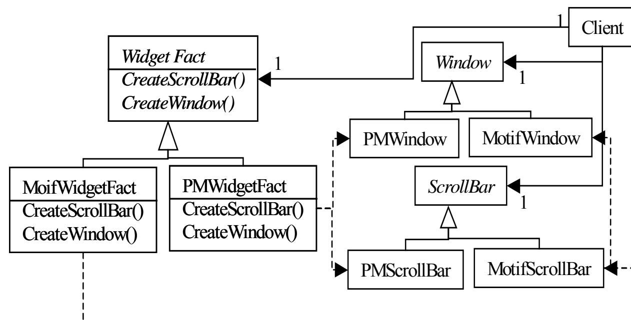

Figure 10 - Application of the Abstract Factory Design Pattern

An informal advice to improve the testability of the class diagram shown Figure 10 is to use as many interfaces as possible for the abstract classes: it avoids links from descendants to parents. If only leave classes in the inheritance hierarchy (classes that have no descendants) are concrete classes (i.e. that may be instantiated), the complexity of the interaction going through this hierarchy is 1. Indeed, if a potential interaction goes though an inheritance hierarchy, and if only leaf classes implement methods, then the interaction only uses one class in the hierarchy and the interaction’s complexity is 1. Figure 11 (b) shows the CDG obtained if all abstract classes are interface classes.

To test the application of the Abstract Factory pattern, we must check if the delegation from the Client to the WIDGETFACTORY creates all objects and does not do anything else. If the design pattern is well applied, the CLIENT class uses creation methods of WINDOW and SCROLLBAR through the “WIDGETFACTORY” inheritance hierarchy and uses other methods directly. This can be specified on the design by adding a «use-consult» on the dependencies from factory classes to the window and scrollbar classes, and a «use-consult» stereotype on the association from CLIENT to WINDOW and SCROLLBAR. In that case there is no interaction at all.

Typical value (example from [Gamma95], Figure 10):

- 4 potential interactions of complexity 4
- with all abstract classes converted to pure interfaces: 4 interactions of complexity 1
- if delegation is well implemented: 0 interaction

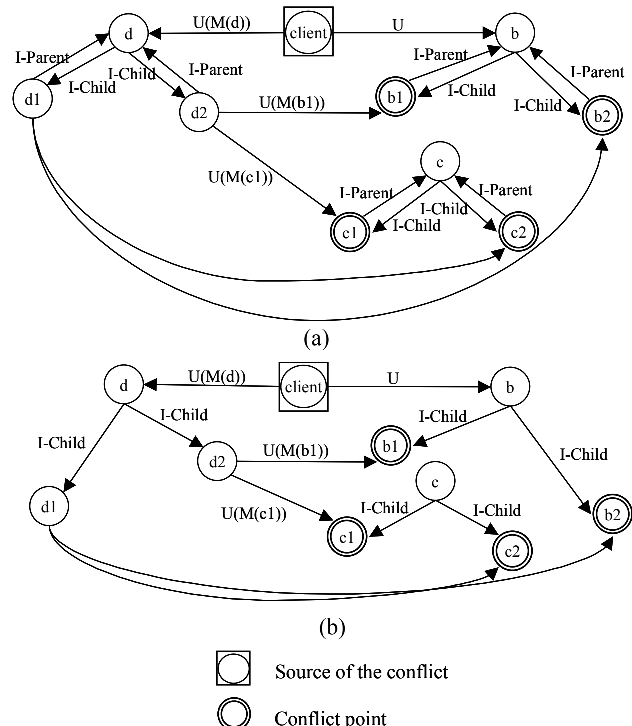

Figure 11 - CDG for the Abstract Factory Design Pattern

Proceedings of the Eighth IEEE Symposium on Software Metrics (METRICS’02)
0-7695-1339-5/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

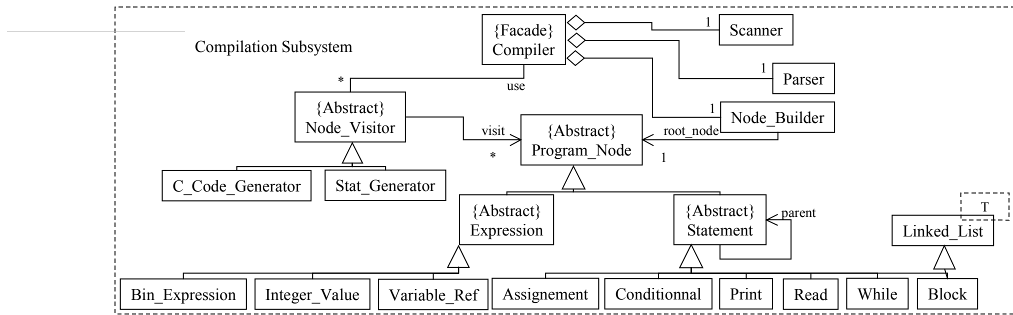

Figure 12 - A compiler architecture

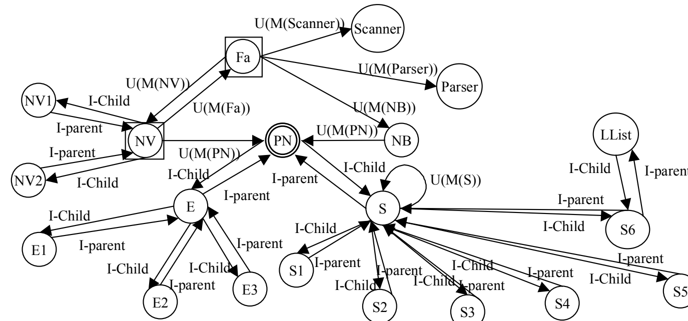

Figure 13 - CDG for the compiler architecture

5.2 A compiler architecture

Figure 12 gives an object-oriented architecture for a compiler taken from [9]. This architecture includes a Scanner class that produces tokens, a Parser that produces an abstract syntax tree using a NODE_BUILDER and a PROGRAM_NODE representing an abstract node in the abstract syntax tree.

A Class Dependency Graph can be derived from this architecture (Figure 13). Two potential class interactions can be detected from this graph. The first one, CI(Fa,PN), is due to the two paths [Fa, NB, PN] and [Fa, NV, PN]. The second potential interaction, CI(NV,PN), is due to the paths [NV, Fa, NB, PN] and [NV, PN]. Both interactions seem quite simple as only four classes, linked by simple uses relationships, are involved. But, their complexity grows enormously because of the eleven classes in the PROGRAM_NODE inheritance hierarchy: 9 descendents-paths of size three are involved in both interactions. The global complexity of this hierarchy is $\sum_{i=1}^{9}(3 \cdot (3 - 1)) = 54$. The NODE_VISITOR inheritance hierarchy has a smaller impact on the complexity since there are only two classes. The complexity for this hierarchy is only 4.

Since all paths involved in the interactions cross the same inheritance hierarchies, they all have the same complexity: 54*4 = 216. In the same way, both interactions have the same complexity that is the product of the two path's complexity: 216*216 = 46656.

Here, the design can be refined with stereotypes on associations from Facade to NODE_VISISTOR and from Facade to NODE_BUILDER. Indeed, Facade instances should use NODE_VISITOR instances only for queries, the association is thus stereotyped «use_consult». The association from Facade to NODE_NUILDER should be stereotyped «use_def» since Facade instances might change the state of NODE_BUILDER instances. If these stereotypes are added to the design, the programmer should not implement any object interactions.

6 Related Work

Testability is at the border of two software research fields. On one hand it is related to testing problems: it evaluates the effort needed to test a piece of software. On the other hand, the testability is a measurement, thus a large part of this work is related to previous work about object-oriented metrics.

Traditionally, testing is divided into three phases, unit testing, integration testing and system testing. This separation is not so clear for testing of an OO system. Due to inheritance and dynamic binding, the control flow of an OO-system is not rooted anymore in the main encapsulation unit, the class. Unit testing, which focuses on classes and methods, cannot capture the interactions distributed throughout the system. The effectiveness of unit testing is thus even more limited to local aspects [10,11] than it is in "traditional" (non-OO) systems. Integration testing, on the other hand, insists more on the component interfaces and on the order in which components are integrated [12,15]. It also may miss some of the interactions among the classes. Finally, at the system level, testing is usually of the "black-box" nature, and often requires, to be really applicable in practice, strong (and possibly unrealistic) assumptions concerning the completeness of behavioral and dynamic models [16]. In this paper, the work we propose is complementary to

IEEE COMPUTER SOCIETY

system testing: it aims at covering object interdependencies with test cases that may be obtained using system testing techniques [17], e.g. derived from use cases and sequence/collaboration diagrams.

Besides, a large number of measures have been proposed to evaluate the quality of object-oriented designs [18], one of them is coupling. The coupling measures the strength of the relationship between two classes. There exist large number of coupling measures, which measure different types of relationships between classes [19].

This paper proposes a mapping of a coupling measurement to precise modeling elements of the UML. The coupling between object (CBO) measure [19,20] corresponds to a set of classes that use each other's. The CBO measure is discussed in terms of testability in [3], and test criteria for this type of relationship among classes are proposed in [5]. These studies focus on each path independently and aim at counting/covering. Here, we concentrate on particular paths that contribute to interactions in the overall system. To our knowledge, this precise contribution to the testability of each dependency participating to coupling has never been studied, and especially in the case of software designed using the UML. To summarize, the goal of the paper is less to limit coupling than to specify roles of links participating to coupling.

7 Conclusion

This paper detailed a testing criterion for object-oriented design to cover object interactions, a model and a testability measurement, and an associated methodology for improving design testability.

The interactions occur when a relationship exists between two objects through several paths. They become more complex if polymorphism due to inheritance is involved. Because, the computation of real objects interactions is not a decidable problem, the notion of class interactions is used. At design level, it provides a computable upper bound for object interactions. Using several features from a UML class diagram, we build a model, the class dependency graph. It detects, counts and evaluates the maximum complexity of class interactions, that is the used testability measurement. We also suggest a set of refinement actions to improve design final testability.

8 Bibliography

1. J.M. Voas and K. Miller, "Software Testability: The New Verification". IEEE Software. Vol.12(3), p. 17-28, 1995.
2. J.M. Voas. "Object-Oriented Software Testability". in proceedings of *International Conference on Achieving Quality in Software*, January 1996.
3. R.V. Binder, "Design for testability in object-oriented systems". Communications of the ACM. Vol.37(9), p. 87-101, 1994.
4. A. Abdurazik and A.J. Offutt. "Using UML collaboration diagrams for static checking and test generation". in proceedings of *UML'00*2000.
5. R.T. Alexander and J. Offutt. "Criteria for Testing Polymorphic Relationships". in proceedings of *ISSRE'00 (Int Symposium on Software Reliability Engineering)*, San Jose, US, October 2000.
6. M. Fowler, "Reducing Coupling". IEEE Software. Vol.18(4), p. 102-104, 2001.
7. S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information". IEEE Transactions on Software Enginnering. Vol.11, p. 367-375, 1985.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software". Professional Computing, Addison-Wesley, 1995.
9. J.-M. Jézéquel, M. Train, and C. Mingins, "Design Patterns and Contracts", Addison-Wesley, 1999.
10. J.-M. Jézéquel, D. Deveaux, and Y. Le Traon, "Reliable Objects: a Lightweight Approach Applied to Java". IEEE Software. Vol.18(4), p. 76-83, 2001.
11. M.J. Harrold and G. Rothermel. "Performing Data Flow Testing on Classes". in proceedings of *FSE (Foundation on Software Engineering)*, New Orleans, US, December 1994.
12. K. Akif, H. Vu Le, Y. Le Traon, and J.-M. Jézéquel. "Selecting an Efficient OO Integration Testing Strategy: An Experimental Comparison of Actual Strategies". in proceedings of *ECOOP (European Conference for Object-Oriented Programming)*, Budapest, Hungary, June 2001.
13. D.C. Kung, J. Gao, P. Hsia, Y. Toyashima, and C. Chen, "On Regression Testing of Object-Oriented Programs". The Journal of Systems and Software. Vol.32(1), p. 21-40, 1996.
14. Y. Le Traon, T. Jéron, J.-M. Jézéquel, and P. Morel, "Efficient OO Integration and Regression Testing". IEEE Transaction on Reliability. Vol.49(1), p. 12-25, 2000.
15. L. Briand and Y. Labiche. "Revisiting Strategies for Ordering Class Integration Testing in the Presence of Dependency Cycles". in proceedings of *ISSRE*, Hong-Kong2001.
16. R.V. Binder, "Testing Object-Oriented Systems: Models, Patterns and Tools", Addison-Wesley, 1999.
17. L. Briand and Y. Labiche. "A UML-based approach to System Testing". in proceedings of *UML*, Toronto2001.
18. ESERG. "Object-Oriented Metrics: an Annotated Bibliography" http://dec.bournemouth.ac.uk/ESERG/bibliography.html
19. L. Briand, J.W. Daly, and J.K. Wüst, "A Unified Framework for Coupling Measurement in Object-Oriented Systems". IEEE Transactions on Software Enginnering. Vol.25(1), p. 91-121, 1999.
20. S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object Oriented Design". IEEE Transactions on Software Enginnering. Vol.20(6), p. 476-493, 1994.

IEEE COMPUTER SOCIETY