



Measuring design testability of a UML class diagram

Benoit Baudry, Yves Le Traon*

IRISA, Campus Universitaire de Beaulieu, 35042 Rennes cedex, France

Received 12 November 2003; revised 19 January 2005; accepted 20 January 2005

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’ and is generalized in the notion of testability anti-pattern: 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. In this paper, the testability measurement we propose counts the number and the complexity of interactions that must be covered during testing. The approach is illustrated on application examples.

© 2005 Published by Elsevier B.V.

Keywords: Object-oriented software measurement; UML; Object-oriented testing; Software design quality; Testability; Anti-patterns

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 complex interactions may appear and lead to difficulties for testing. Testability, informally defined as the easiness to test a piece of software, is a strongly desired feature of software. It tends to make the validation phase more efficient in exposing faults during testing, and consequently to increase quality of the end-product for clients’ satisfaction. Furthermore, testability is a criterion of crucial importance to software developers since the sooner it can be estimated, the better the software architecture will be organized to improve subsequent implementation and maintenance. This question of testability [1] has been revived with the object-orientation [2].

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 [3], 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.

Like for any classical software, the difficulty for testing is due to the existence of client/supplier relationships in

* Corresponding author.

E-mail addresses: benoit.baudry@irisa.fr (B. Baudry), yves.le_traon@irisa.fr (Y.L. Traon).

113 the system. Indeed, if there were no client in the software
 114 there would be no defined set of executions and thus nothing
 115 to test. Thus, after unit testing, failures should only occur
 116 because of a misuse due to wrong interactions between
 117 objects: these interactions go throughout the architecture
 118 and are made more complex if the client/supplier dependen-
 119 cies traverse inheritance trees. Polymorphic dependen-
 120 cies multiply the number of potential object types that may
 121 interact with various—and possibly false—implemen-
 122 tations. This paper introduces a testing criterion that
 123 requires the coverage of these object interactions. To be
 124 realistically applied, the number of test cases must be
 125 reasonable and the paper proposes an estimate of the testing
 126 effort, measured by approximating the number of object
 127 interactions from the UML class diagram.

128 The number of object interactions is estimated by the
 129 number of ‘class interactions’: a class interaction is a
 130 topological configuration that occurs if a class is supplier
 131 from another through various possible paths of dependencies.
 132 Moreover, this work proposes a complexity measurement for
 133 class interactions based on the complexity of inheritance
 134 hierarchies that are present along in the paths of dependencies
 135 involved in the interaction. The number of class interactions
 136 being an estimate of object interactions (it is actually the
 137 maximum number of possible object interactions), it is also an
 138 estimate of the number of test cases that will have to be
 139 exhibited to test the system. Moreover, the complexity of class
 140 interactions is an estimate of the complexity of producing the
 141 test cases. The number and complexity of class interactions is
 142 thus an estimate of the difficulty of testing a system, and it is
 143 the testability measurement proposed in this paper.

144 The proposed testability measurement is computed from
 145 the class diagram and thus offers a worst-case estimate of the
 146 testability of the implementation (the case where each class
 147 interaction is actually implemented), which can be different
 148 from the actual testability. However, we believe this is still
 149 useful from a methodological point view, since the class
 150 interactions are still specific points the designer should be
 151 aware of in terms of testability (it is thus useful to identify
 152 them automatically). Moreover, associating a complexity
 153 measure to these interactions enables the designer to focus on
 154 the most complex to improve the design.

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

- 157 – to provide a model to capture class interactions and
- 158 pinpoint classes that cause the interactions,
- 159 – to identify hard-to test interactions and measure their
- 160 number and complexity due to polymorphic uses,
- 161 considered here as our estimate of design testability,
- 162 – to suggest improvements on the design to reduce the
- 163 number and complexity of class interactions: these
- 164 improvements at design level are realistic since static
- 165 verifications on the code ensure their implementation,
- 166 – *in fine* to provide a way of accepting or rejecting a design
- 167 based on testability analysis. The design is rejected when
- 168

no improvement can be added to limit the object
 interactions.

The measure of testability we propose is a counting
 measure: it counts the number and the complexity of class
 interactions to be covered by test cases. It is thus a global test
 cost measure for OO systems designed with a class diagram.

Section 2 opens with a general presentation of the
 testability measurement, and what particular points should
 be studied in an object-oriented context. It also introduces a
 methodology to design testable OO systems. Section 3
 analyses the notion of testability anti-pattern, and proposes
 precise definitions in terms of elements of a UML class
 diagram. Then, it defines a testing criterion and illustrates
 the test generation on a small example. Section 4 defines a
 graph model that can be derived from a class diagram and
 from which all anti-patterns can be detected. We also
 propose a measure for the complexity of anti-patterns, that
 can be automatically computed from the graph model.
 Section 5 gives clues for refinements that can drive the
 design closer to the implementation and thus make the
 measurement on the design closer to actual testability
 problems. Section 6 gives two application examples, and
 Section 7 summarizes related work.

2. Testability of OO design: definitions and methodology

This section introduces the context for this work. It starts
 with definitions about software testability and what type of
 information the measure must capture. Then, we describe
 specific testing problems that appear in many OO
 architectures. This leads to a proposal for a methodology
 for testability of OO software. At last, we present an
 example that is used through the paper for illustration.

2.1. Software testability

In this paper, testability serves two goals: a technical one,
 and a more managerial one. Testability is a tool for the
 software designer who wishes to identify hard-to-test
 systems while still at the design stage. For the project
 manager, they provide a means to decide whether a trade-off
 in terms of cost is worth searching for between solutions
 based on different designs and different testing methods.

For example, whatever the quality factor under scrutiny
 is, the type of situations that a designer might hope to
 identify include:

- stress points where there is a bad degree of this factor,
 and thus a need to improve the design,
- inadequate refinement leading to a sharp and undesired
 variation in this factor.

Our general definition for testability is the following.

Definition. Testability. We define the quality factor *testability* as the ease of testing a piece of software design. This easiness is both an intrinsic property of the design (thus a proper characteristic of the product) and a property correlated to the testing strategy which is used to reach a chosen test criterion (thus, a joint characteristic of the product and the process). Testability is influenced by three parameters:

- Global test cost: the overall test cost to reach a (a joint product-process property),
- Controllability: the overall easiness of generating test data (an intrinsic software property),
- Observability: the overall easiness of checking the validity of the execution results (another intrinsic software property).

In this paper, we focus on a global test cost measure of testability.

Definition. Global test cost. This factor concerns the testing effort needed to reach a given testing criterion. It relates to the size of the test set, the difficulty of generating the test data to reach a given test adequacy criterion and the difficulty of deciding on the validity of the run results.

The test objective is not only to cover or execute each part of the model but also to reveal hidden faults. From this viewpoint, the notions of controllability and observability of a software component, introduced by Freedman [4] are complementary to the global test cost. For instance, controllability is related to the effective coverage of the declared output domain from the input domain. However, his approach fails at considering the inherent difficulty to execute and infect a component and propagate the faulty state to the outputs. This drawback has been well analyzed by Voas’ pragmatic approach [1,5], at code level. At design stage, several measurements have been proposed to estimate an information loss, such as Voas’ Domain/Range Ratio (Drr) [6] for imperative programs or the controllability/observability measurements stated in [7,8] for data flow designs. Weide et al. [9] have characterized observability and controllability on abstract data types from an understandability viewpoint for software reuse. Up to now, the question of measuring the controllability/observability in the OO context at design stage has not found any satisfying answer. We do not address this issue in this paper. Concerning the ‘global test cost’ category of measures, Bieman and Schultz

[10] have examined the number of test cases that are needed to satisfy the all-du-paths criterion [11], and in catalogues of measures such as [12], testability is indirectly estimated, based on the general assumption that testability is likely to degrade with a more highly coupled system of objects.

To obtain a relevant testability measure, specific OO issues must be taken into account: the control distributed all over the architecture and the numerous and complex interactions among objects (due to dynamic binding and polymorphism). The literature insists on the difficulty to elaborate valid measurements [13–16], and we can easily find catalogues of measures, typically counting every attributes that can be found in an object-oriented system (number of methods, depth of inheritance trees, etc.). The measures are obtained neither from the observation of case studies nor by a clear intuitive relation between the factor under measurement and the measured attributes of the software. In this paper, since the measured factor, the testability, first appear as quite abstract and unclear, we choose to have a pragmatic approach. Conversely to our previous research based on axiomatization [8,17], the measurement philosophy is thus different from classical ‘top down’ approaches. We also renounce to cover all the spectrum of what may be measured and related ‘a posteriori’ to testability. Our methodology is ‘bottom-up’, in the sense we first studied concrete applications carefully, in order to identify the attributes that impact the testability, in a precise testing context. To do that, we need to define precisely the testing task (testing criterion that has to be satisfied) and then be able to evaluate the effort to test a piece of software according to this criterion (evaluate all the interactions that have to be covered as well as their complexity). The concept we identify as relevant of a ‘testability weakness’ is called a *testability anti-pattern* and the measured attribute a *class interaction* or *self-usage interaction*. The global test cost measure we define is equal to the number of detected testability anti-patterns. It will be defined precisely in the following after the study of an application example.

2.2. Designing for testability: a methodology

Fig. 1 summarizes a methodology that helps improve a design’s testability. The main specification for the testability analysis is the class diagram. The first step of the proposed method consists in running a testability analysis on the class diagram. This analysis detects points

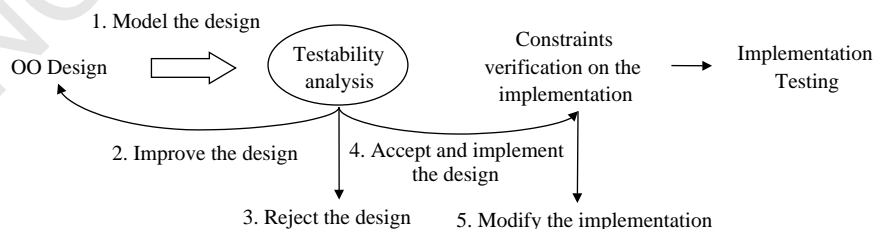


Fig. 1. Improving testability of OO designs.

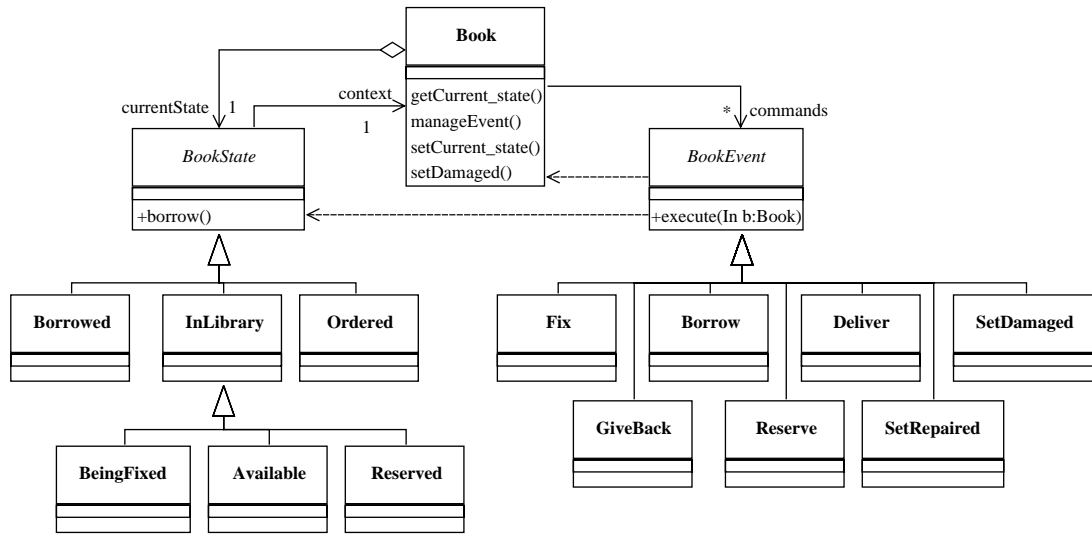


Fig. 2. UML class diagram for book manager sub-system.

in the design that have to be improved for testability. As we will see in Section 2.3 these points correspond to particular configurations in the diagram that can lead to hard-to-test implementations. To run this analysis automatically on a class diagram, we need a model that can be derived from the diagram and from which it is possible to detect hard points for testability in an unambiguous way.

As a result, the testability analysis lists all the points that need to be improved in the design. As we will see in Section 4, it also associates a complexity measure to these points. Once the analysis has been run, it is possible to improve the design at those specific points, or to reject as too difficult to test, or to accept this design as testable and implement it. The design can be improved, either by reducing coupling in the architecture [18], or by expressing constraints that will help the developer avoid implementing error-prone object interactions. Our suggestion (Section 5) is to use dedicated

stereotypes on association and dependencies specifying more clearly the type of usage that must be implemented (creation, reading...). So, when the design is implemented, the constraints are checked, and the implementation may need to be modified if the constraints are not verified.

2.3. Example

We introduce here a UML class diagram that serves as an illustration example all along this paper. This diagram corresponds to a sub-system in charge of managing books in a larger library system (Fig. 2). All the classes are given but we show only the methods that are used to illustrate particular points in the following sections. This class diagram is the design for a system implementing the UML statechart presented Fig. 3. The statechart describes the dynamic behavior of a book object. An object is created

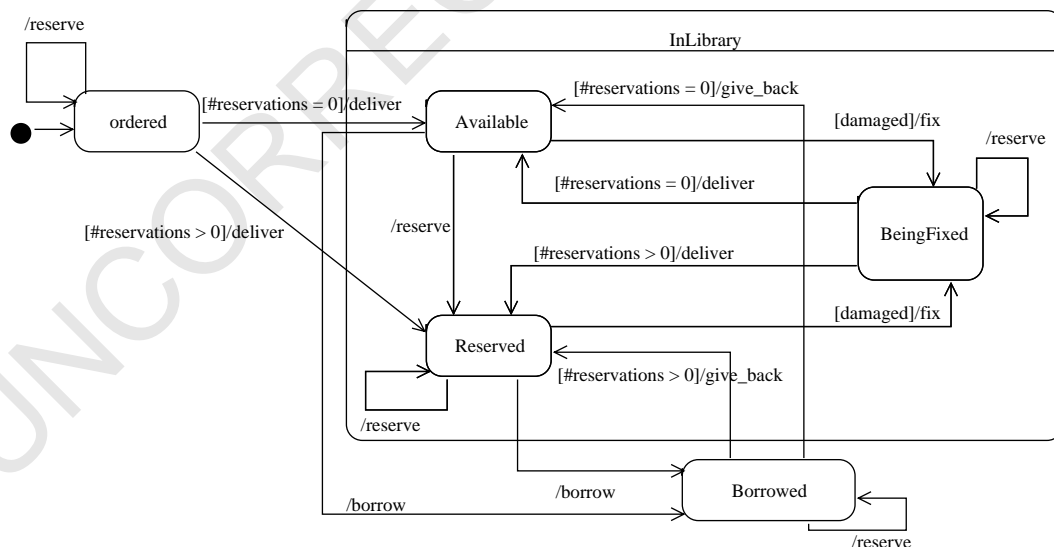


Fig. 3. UML statechart for book management.

when a book has been ordered (initial state). Once the book is ordered, it can be reserved at any time. When it comes in the library, it is either available or reserved, and it can then be borrowed. If the book is damaged and is in the library, it can be fixed. The design proposed to implement this statechart (Fig. 2) is based on two design patterns [19]: the state pattern that reifies each state of the statechart in one class and the command pattern that reifies events.

Now that the general context for this work has been presented, Section 3 details the particular interactions we focus on, as testability weaknesses in a class diagram. Then, we propose a testing adequacy criterion to cover these interactions and illustrate this criterion by writing test cases for the book manager sub-system.

3. Test criterion and testability anti-patterns for OO systems

A testing criterion is needed to detect object misuses due to erroneous interactions. Here we propose a criterion based on the UML as a reference specification, that aims at covering all object-to-object dependencies that should be tested. 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. Once the testability problems have been highlighted, a design can be either improved or rejected as not testable.

This section starts with an informal analysis of testability problems of the book manager design (Fig. 2). These problems actually correspond to particular configurations that can be found in a class diagram and lead to hard-to-test implementations. These configurations are called *testability anti-patterns*, as they describe patterns that should be avoided for a testable design. We also explain how inheritance can increase the complexity of anti-patterns.

After that, those anti-patterns are defined more precisely in terms of elements in a UML class diagram. Based on these definitions, we are able to express a testing criterion to cover those interactions. The section ends with the generation of test cases for the book manager that verify the testing criterion.

3.1. Informal analysis of testability anti-patterns

This section aims at pointing, in an informal way, interactions in a class diagram that can lead to problems for testing the corresponding implementation. We look at the class diagram given in Fig. 2 as an example. This architecture is a typical object-oriented design. It uses basic constructs of object-orientation: inheritance, abstract classes, associations, aggregation 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 BOOK and BOOKSTATE are interdependent. This type of architecture has a considerable potential for faulty behavior. For example, BOOKEVENT may depend on BOOK 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 identified in order to estimate the verification and validation effort. The two potential sources of problems are the following:

- When a method `m1` in class `BOOK` uses a method `m` of class `BOOKSTATE`, the class `BOOKSTATE` may use `BOOK` to process `m`. That means that the class `BOOK` might use itself when it uses `BOOKSTATE` to process part of its work.
- When a class of `BOOKEVENT` uses `BOOK`, it might do so in two different ways: directly by declaring an instance of class `BOOK`, or through a use of `BOOKSTATE` which uses `BOOK`.

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 weaknesses for testability: interactions from one class to another we call *class interactions*, and a configuration we call *self-usage* that corresponds to a class that uses itself by transitive usage dependencies. We call these weaknesses *testability anti-patterns*. An anti-pattern describes a solution to a recurrent problem that generates negative consequences to a project [20]. As design patterns, anti-patterns can be described with the following general format: the main causes of its occurrence, the symptoms describing ways to recognize its presence, the consequences that may result from this bad solution, and what should be done to transform it into a better solution.

Testability anti-pattern. A testability anti-pattern is a design solution that presents a configuration in the class diagram which increases the testing effort.

In this paper the testing effort is estimated by the number of test cases as well as the complexity to produce the test cases needed to verify a given test criterion. An anti-pattern is thus a design decision that increases the number and/or the complexity of test cases. Two specific configurations in a class diagram have been identified as such design decisions: class interactions and self-usage. Both designs present hard points for testing because in both cases, test cases must be generated to cover paths that go through several classes. In most cases if the path is actually coverable, the test data is very specific and thus difficult to generate. Moreover, if several paths are involved in class interactions or self-usages, test cases must check the combinations of those different paths, which also increases the necessary effort to produce the test cases. For these reasons class interactions and self-usages that are identified on the class diagram are testability anti-patterns.

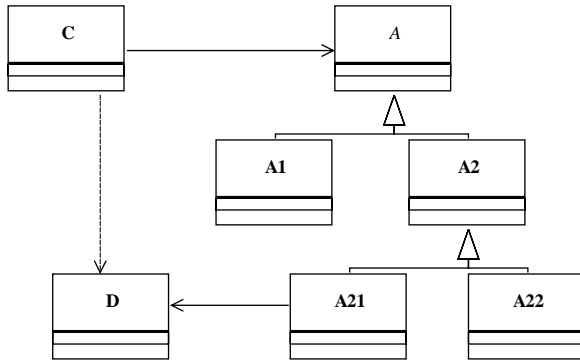


Fig. 4. Concurrent usage through an inheritance hierarchy.

The complexity of both anti-patterns worsen when usage dependencies go through an inheritance tree because of polymorphism. Section 3.2 illustrates this point.

3.2. Inheritance complexity

The complexity due to inheritance appears when transitive dependencies go through one or several inheritance hierarchies. This section aims at giving the intuition of the complexity of polymorphic relationships, based on the class diagram of Fig. 4. The figure presents 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 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.

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.

3.3. Test criterion for UML class diagrams

In this section, we come back on the anti-patterns that have been identified in Section 3.1 and define them precisely in terms of elements in a UML class diagram. Then, we define a test criterion that requires the coverage of those anti-patterns when testing the implementation. This testing criterion concentrates on the hard-to-detect errors that can appear when side effects may occur, i.e. when one or several objects may modify the state of an object using independent paths of dependencies. Such combinations of dependencies can lead to inconsistent states for the handled objects.

In an OO system, classes depend on each other's for their processing. A class A is said to use a class B if methods in

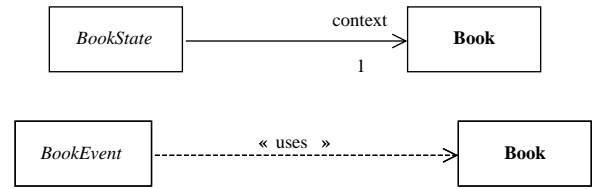


Fig. 5. Association or dependency between classes.

A call methods from B, either through an attribute or a local variable of type B. The UML allows the designer to illustrate this relationship on a class diagram drawing either an association between the classes or a dependency stereotyped «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. Fig. 5 illustrates the two types of UML dependencies: an association between BOOKSTATE and BOOK classes and dependency BOOKEVENT and BOOK classes.

The direct usage relationship can be extended to the *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; this is due to transitive relationships.

Transitive usage relationship. Direct usage relationships are considered transitive. This means that, if there is a direct usage relationship exist from class A to class B and from B to C then, there is a transitive usage relationship from A to C called A R C.

There may be several transitive usage relationships from A to C, in that case the *i*th transitive usage relationship from A to C is denoted $A R_i B$. 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.

For example, Fig. 6 illustrates two relationships between classes BOOKEVENT and BOOK. A «uses» dependency

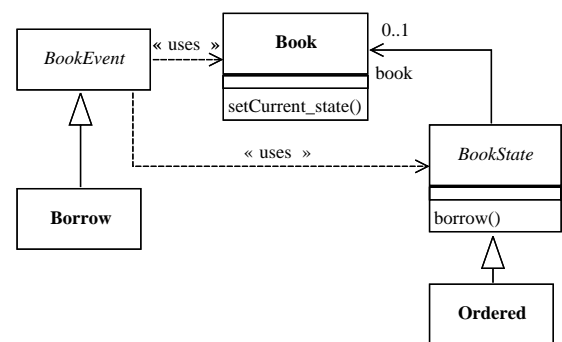


Fig. 6. Transitive relationship between BookEvent and Book through BookState.

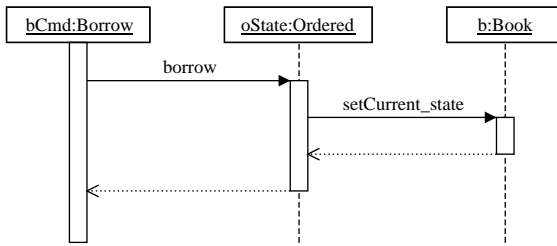


Fig. 7. Sequence diagram illustrating a real indirect relationship between BookEvent and Book.

between the two classes specifies a direct usage relationship. The second relationship is a transitive one through the BOOKSTATE class. The BOOKEVENT class depends on the BOOKSTATE class which depends on BOOK. Thus BOOKEVENT may depend transitively on BOOK when calling services from BOOKSTATE. This relationship is a real relationship if methods of BOOKSTATE, called by BOOKEVENT objects, use services from BOOK. The sequence diagram from Fig. 7 illustrates a real transitive relationship between BOOKEVENT and BOOK. When a BORROW object (of type BOOKEVENT) calls the borrow() method of class ORDERED, this method calls the setCurrent_state() method of BOOK. Thus, a BORROW object actually depends on a BOOK object through a BOOKSTATE object.

Let us define now the notions of *class interaction* and *self-usage interaction*. These interactions are potential interactions since they are 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 be worsen when the design evolves and is implemented. We thus also define *object interactions*, which are real interactions since relationships between running objects are involved. Some of them can be detected at the design level from UML 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 interaction in the system. Those two notions are made more formal in the following definitions.

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

- $\exists i$ and $j, i \neq j$, such as $A R_i B$ and $A R_j B$,
- A self-usage interaction occurs around class A iff: $A R_i A$

Fig. 6 illustrates a class interaction between BookEvent and Book. This interaction involves two dependencies between those two classes. More generally, 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:

- $\exists i$ and $j, i \neq j$, such as $A R_i B$ and $A R_j B$, or $A R_i A$
- R_i and R_j are real transitive relationships for o_1 and o_2 .

For example, if the sequence diagram of Fig. 7 is associated to the class diagram of Fig. 6, the class interaction between the BOOKEVENT and BOOK classes is also an object interaction.

Property. The number of class interactions and self-usage 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.

Now that we have defined the class and object interactions, we can give our testing criterion.

Test criterion. For each class interaction, either a test case is produced that exhibits a corresponding object interaction, either 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 statically typed. Since the number of class interactions is an upper bound of the number of object interactions, we recommend to put additional information on the design that would reduce the number of class interactions. These additional pieces of information are design constraints for the programmer (e.g. expressed using UML stereotypes): one can statically verify that the implementation fits the constraints. This means that using static verification at the code level reduces the testing effort. As an example, being given a «instantiate» 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.

3.4. Example for test generation

This section introduces an example for test generation process using the adequacy criterion defined in Section 3.3. The example is based on the class diagram of Fig. 3. First, testability anti-patterns are identified in the diagram, then test cases are produced when interactions are implemented as actual object interactions.

Two self-usage interactions and one class interaction appear on the class diagram of Fig. 3:

- SU1 from BOOK to itself through BOOKSTATE
- SU2 from BOOK through BOOKEVENT
- CI between BOOKEVENT and BOOK through two different paths (a direct one and a path going through BOOKSTATE).

The testing criterion states that a test case has to be produced for each class or self-usage interactions to exhibit an actual object interaction. For potential interactions that

785 are not implemented as object interactions, a report stating
786 this absence of actual interaction has to be produced.

787 The entry point to test this set of classes is the `BOOK` class.
788 Thus, a test case consists in creating a `BOOK` instance and
789 calling methods on this object. If the reader wants to check
790 the source code of the example, it is available at the
791 following URL: <http://www.irisa.fr/triskell/results/Book/>.

792
793 **3.4.1. SU1 interaction**

794 Our first test objective is the self-usage interaction going
795 from `BOOK` to itself through the `BOOKEVENT` class (SU1). To
796 cover this interaction, the test case has to call a method in
797 `BOOK` that uses the `commands` set, and this method has to
798 call a method in the `BOOKEVENT` class that uses the `BOOK`. In
799 the `BOOK` class, only the `manageEvent()` method uses
800 `commands`. In all the concrete event classes, the methods
801 are of the following form:

```
802     execute(Book b) {...}
```

803
804 Thus, a test case that calls the `manageEvent()` -
805 method in `BOOK`, covers the interaction. Here is an example
806 of such a test case (TC1):

```
807     public void testManageEvent() {  
808         Book b=new Book();  
809         b.manageEvent("setDamaged");  
810     }  
811  
812
```

813
814 **3.4.2. SU2 interaction**

815 The second test objective is the interaction going from
816 `BOOK` to itself through the `BOOKSTATE` class (SU2). A test
817 case covering this interaction should call a method that uses
818 the `currentState` attribute in `BOOK`. Actually, there is
819 no such method in the `Book` class, this attribute is only read
820 by the `getState()` method. The self-usage interaction
821 we are trying to test has thus not been transformed in an
822 object interaction in the implementation. Since there is no
823 actual self-usage interaction in the implementation, no test
824 case needs to be defined to cover SU2.

825 This example illustrates the fact that class interactions
826 are a worst-case estimation of the testing effort for the
827 implementation corresponding to a class diagram. Indeed,
828 some interactions detected on the class diagram (and thus
829 identified as hard-points for testing on the design) are not
830 implemented as interactions between objects and are not
831 taken into account for testing the implementation (and are
832 not taken into account in the testing effort).

833
834 **3.4.3. CI interaction**

835 The third objective is to exhibit an object interaction
836 between `BOOKEVENT` and `Book` through two different paths
837 (CI). Since the `BOOK` class is the entry point for testing,
838 the test case has to call a method that uses the `commands` set.
839 When writing a test case for the first test objective, we have
840 seen that a call to the `manageEvent()` method covers

841 the relationship from `BOOK` to `BOOKEVENT`, and also the one
842 from `BOOKEVENT` to `BOOK`. Thus the direct path from
843 `BOOKEVENT` to `BOOK` is covered by test case calling
844 `manageEvent()` in the `BOOK` class. To cover the second
845 path from `BOOKEVENT` to `BOOK` (through `BOOKSTATE`), the
846 call to `manageEvent()` has to cover the relationship
847 between `BOOKEVENT` and `BOOKSTATE`. This can be done by
848 calling an event which processing depends on the actual
849 state of the `BOOK` instance. In that case the `execute()`
850 method in the concrete events has the following form:

```
851     execute(Book b) {b.getState();...}
```

852 Then, if a transition in the statechart is triggered by the
853 called event, then the relationship between `BOOKSTATE` and
854 `BOOK` is covered, since in that case the method in the
855 concrete state calls a method on the context attribute. For
856 example, the `borrow()` method in the `AVAILABLE` class
857 has to change the state of the context to which it is
858 associated, since the `borrow` event in the `AVAILABLE` state
859 triggers a transition from `Available` to the `BORROWED` state.
860 Here is the corresponding code:

```
861     class Available{  
862         public void borrow(){context.chan-  
863             geState(new Borrowed());}  
864     }  
865
```

866 To summarize this third test objective, the following test
867 case covers the interaction between `BOOKEVENT` and `BOOK`
868 through two different paths and Fig. 8 gives the sequence
869 diagram for this test case (TC2).
870

```
871     public void testManageEvent() {  
872         Book b=new Book(); //the book is in the  
873         ordered state  
874         b.manageEvent('deliver'); //puts the  
875         book in the available state  
876         b.manageEvent('borrow');  
877     }  
878
```

879 The Table 1 summarizes the results for testing an
880 implementation of the book manager system (Fig. 2)
881 according to the test criterion of Section 3.4. This table
882

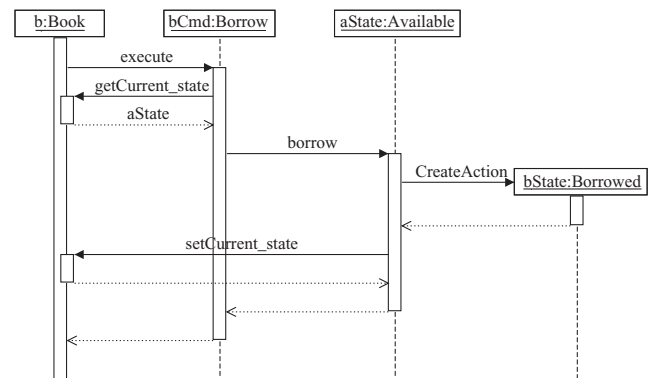


Fig. 8. Sequence diagram for test case n°3.

Table 1
Test report for the book manager sub-system

	SU1	SU2	CI
Status		Infeasible	
TC1	X		
TC2			X

presents the status for each anti-patterns detected in the system (feasible or not), then, for each test case, which anti-pattern is covered. Actually, there are much more than three interactions that should be tested due to the fact that BOOKSTATE and BOOKEVENT have many sub-classes. Section 4.3, details a way to compute the complexity of these interactions. This complexity corresponds to the maximum number of interactions that can appear in presence of inheritance, and that have to be tested.

We have developed a tool that can help generating test cases that satisfy the test criterion. This tool is called JTracor and is available at <http://franck.fleurey.free.fr/JTracor/index.htm>. It produces execution traces for java programs. This tool enables to know which objects have actually interacted, and which methods have been called by those objects. The traces obtained when running TC1 and TC2 are given in Appendix B.

4. Modeling testability anti-patterns

In this section, we describe rules for building a graph to capture testing interactions from an object-oriented system described with the UML. Definitions are needed about this graph, called Class Dependency Graph. Then, topological rules on the graph are given that formally determine potential interactions. It serves as a basis for applying classical graph algorithms to detect interactions and measure their complexity.

4.1. Graph construction from a UML model

This section provides several definitions about the class dependency graph model. The graph is an oriented labeled

graph, the following thus defines the various labels that can be found in the graph. Moreover, the definitions provide information on the way the graph is derived from a UML class diagram.

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

Definition. *Class dependency graph (CDG).* A class dependency graph is a pair $CDG = (X, \Gamma)$, where

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

Γ is a 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 two classes. An edge is labeled by the type of dependency that exists between the classes, namely usage dependencies and inheritance.

Remark. 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 .

Definition. *Edge labels.* Every edge in a CDG represents a dependency between two classes of an object-oriented system. Let $c \in C, d \in C$, the edge between vertices c and d is labeled by the type of dependency that exists between c and d . Dependencies can be of two types: usage(labeled U) if c uses d , or inheritance(labeled I) if $c \neq d$ and c inherits from d . Both labels, carry extra information.

Definition. *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 . For this set of method, the default value is $M(d)$ (as long as we do not know the sub-set of $M(d)$ used by c). This transformation is illustrated Fig. 9 (a).

Remark. In the case of Usage dependency named «instantiate» or «create» between classes C and D, the set of methods associated to the label U would be $(createD())$ indicating that C only calls the creation method of class D through this usage relationship.

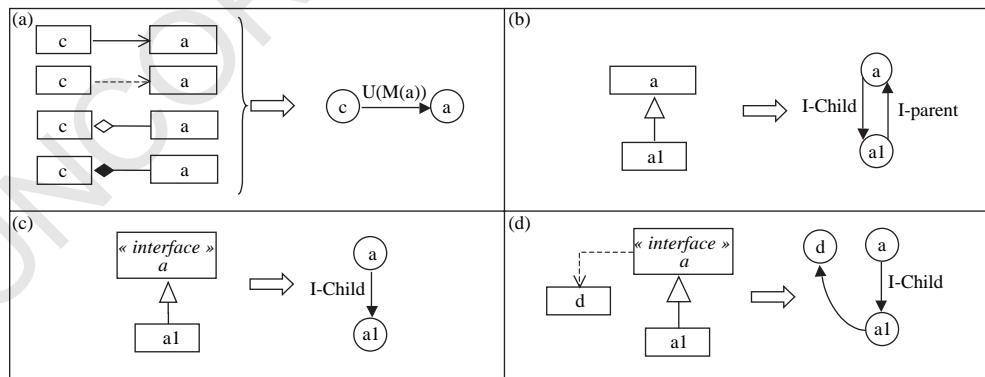


Fig. 9. Basic transformations from a UML class diagram to a CDG.

1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064

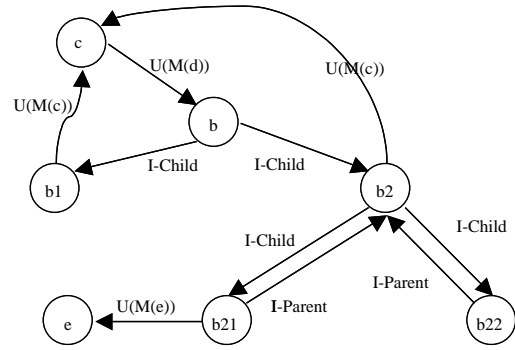
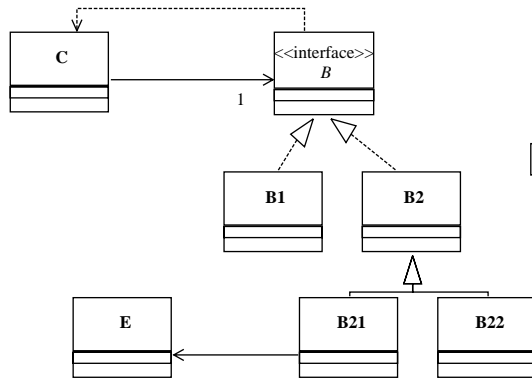


Fig. 10. CDG example.

Definition. Label I. The inheritance label is specialized in two labels (Fig. 9(b)). Let $c \in C$, $d \in C - \{c\}$, if $d \in \text{Parent}(c)$:

- There is an edge (d,c) labeled I—child. 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.
- There is an edge (c,d) labeled I—parent. From a testing point of view, this dependency from the child to the parent is obvious: c uses d when it calls a method $m \in M_{\text{INH}}(c)$.

About the definition of label I, it has to be noticed that, in the case of pure interfaces, there is only one edge going from the interface to its subclasses. Indeed, the subclasses do not depend on the super class since this one is empty (pure interface). However, the edge from the interface to its subclasses is still meaningful to indicate the dependence between the interface and the classes that implement the services it defines (in that way, the graph reflects that a client of the interface actually depends on the subclasses that implement the services, by transitivity).

Example: Fig. 10, shows a class dependency graph obtained from a small class diagram, by applying transformation rules given in the definitions above.

4.2. Detecting testability anti-patterns from the CDG

In this section, we come back on the anti-patterns informally described in Section 3.1, and give more precise definitions of these in terms of the CDG model. First we recall the definitions of paths and cycles in graphs, then the class interaction and the self-usage configurations are defined formally using the graph model.

Definition. Path. A path P in a CDG is a sequence of vertices $P = [x_{i1}, x_{i2}, x_{i3}, \dots, x_{ik}]$, such that:

$$(x_{i1}, x_{i2}) \in \Gamma, (x_{i2}, x_{i3}) \in \Gamma, \dots, (x_{ik-1}, x_{ik}) \in \Gamma$$

x_{i1} is the origin of the path and is called $\text{origin}(P)$

x_{ik} the end and is called $\text{end}(P)$

the x_{ij} ($2 \leq j \leq k-1$), are the intermediate vertices (we call the set of intermediate vertices $\text{itVertices}(P)$).

Definition. Cycle. Let P be a path, P is a cycle if and only if $\text{end}(P) = \text{origin}(P)$.

Definition. 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 Fig. 10, $[c, b, b2, b22]$ or $[c, b, b2, b21, e]$ are elementary paths, but $[c, b, b2, b21, b2]$ is not. In the same way, $[b, b1, b]$ is an elementary cycle, but $[b, b2, b21, b2, b]$ is not.

Definition. Class interaction (CI). There exists a class interaction from class $c \in C$ to class $d \in C - \{c\}$ ($CI(c,d)$) if \exists at least two elementary paths P_1 and P_2 , $P_1 \neq P_2$ such that:

$$(\text{origin}(P_1) = \text{origin}(P_2) = c) \wedge (\text{end}(P_1) = \text{end}(P_2))$$

$$= d \wedge (\text{itVertices}(P_1) \neq \text{itVertices}(P_2)).$$

There is one constraint about paths involved in the class interaction. For a path going through an inheritance hierarchy, it 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.

On Fig. 11, 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.

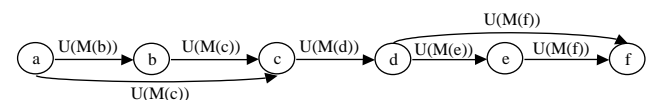


Fig. 11. CI on a CDG.

1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120

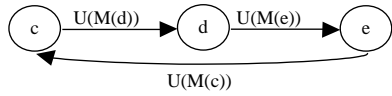


Fig. 12. SU on a CDG.

This definition of the CI interaction, takes into account only unitary interactions: on CDG of Fig. 11, only two potential interactions are detected: $CI(a,c)$ and $CI(d,f)$, a bigger interaction which could be $CI(a,f)$ is not detected. We assume that detecting only unitary interactions is sufficient, because if interactions $CI(a,c)$ and $CI(d,f)$ are solved, the bigger interaction $CI(a,f)$ is also solved.

Definition. Self usage (SU). There exists a self usage on class $c \in C$ ($SU(c)$), if there exists an elementary cycle which origin is c .

There is one constraint about the cycle, if it goes through an inheritance hierarchy, it 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

Fig. 12 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.

4.3. Measuring the complexity of anti-patterns

The complexity of an anti-pattern can now be formalized by taking into account polymorphism in the system. This complexity increases when one or several paths involved goes through a strongly connected component (SCC) of the graph corresponding to an inheritance hierarchy. This increase is due to the fact that the classes in an inheritance hierarchy interact with their ancestor and children classes. So when there is a class C, that is part of an inheritance tree, along a path involved in an anti-pattern, all the classes in the anti-pattern interact with C and the ancestor and children of C.

The complexity measure we detail here aims at computing the exact maximum number of interactions involved in an anti-pattern, in presence of inheritance trees. As it was stated

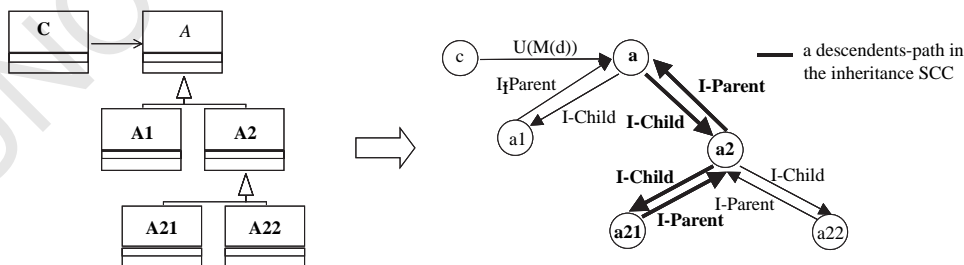


Fig. 13. Slice in a SCC corresponding to an inheritance hierarchy.

in the introduction of this paper, the proposed complexity measurement is based on the number of paths involved in one anti-pattern as well as on the complexity due to the inheritance hierarchy that are crossed by these paths. This gives a good feedback on the testability of the class diagram since the more paths are involved the more test cases have to be written. Moreover, the longer is one path, and the more inheritance hierarchies it traverses, the more difficult it is to write a test case. At last, each path in one interaction (each behavior) should be tested in combination with each other. Indeed, a test case that covers one path checks the consistency of the target class for the source class when using this paths, but the test must also check the consistency of the target class when the paths are combined. So the complexity of an interaction is the combination of the complexities of all paths in the interaction.

Definition. Complexity of interaction. Let $P_1, \dots, P_{nbPaths}$ be $nbPaths$ different paths corresponding to a class interaction CI. The complexity of the interaction is linked to the complexity of the different paths in the following way:

complexity(CI)

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

The complexity of a path is defined in the following.

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

As defined earlier, paths involved in an interaction have can go through an inheritance hierarchy only in one direction. So, a sub-component can be extracted from a SCC. This sub-component corresponds to a slice of the inheritance hierarchy going from a root class to a leaf as shown Fig. 13. This sub-component is called a *descendants-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 descendants-path, which are not pure interfaces, the complexity of the sub-component is $n(n-1)$, because

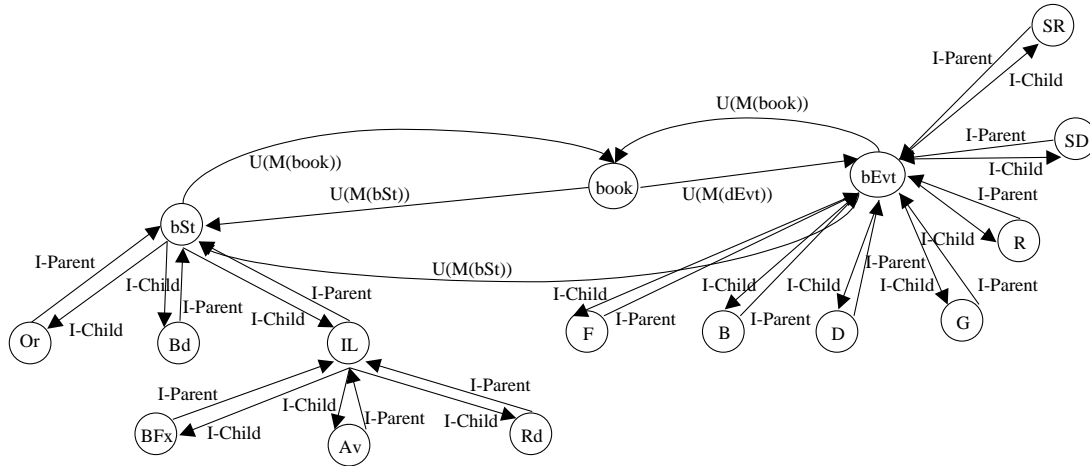


Fig. 14. CDG for the book manager system.

every class has a relationship with each of the $(n - 1)$ others: $n \cdot (n - 1)$ interactions may occur that must be tested.

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 a hierarchy can have a relationship with every class of the other hierarchy.

Definition. Complexity of a path in a class interaction. Let P be a path involved in a class interaction, $IH_1, \dots, IH_{nbCrossed}$ be $nbCrossed$ inheritance hierarchies crossed by P

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

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 Fig. 14, the path [bEvt, bSt, book] goes through the root class of the BOOKSTATE inheritance hierarchy. Since BOOKSTATE is not a leaf in the inheritance hierarchy, all the descendents-paths starting with the node bSt have to be taken into account for the computation of the complexity of the path [bEvt, bSt, book]. The descendents-paths [bSt, Or], [bSt, Bd], [bSt, IL, BFX], [bSt, IL, Av], [bSt, IL, Rd] are involved in the complexity of the path [bEvt, bSt, book].

Definition. 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, \dots, dp_{nbDP} , the $nbDP$ descendents-path in IH influencing P 's complexity.

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

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.

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

$$complexity(dp) = h(h - 1)$$

The testability measure is being implemented as an external component of the industrial CASE tool Objecteering (www.objecteering.com). It is an object oriented CASE tool created by the French firm Softeam. Objecteering/UML modeler covers all UML models and can be used to model entire applications from analysis to semi-automatic code generation. Section 4.4 gives an example for the complexity measurement.

4.4. Measuring the complexity of the book manager system

Fig. 14, gives the class dependency graph for the class diagram of Fig. 2. We detail here the computation for the complexity of the class interaction from Bookevent to Book. The complexity is the product of the complexities of the two different paths involved in this interaction. The first path P1 is a direct link from bEvt to book, the second path P2 is a path from bEvt to book going through bSt.

Even if P1 is a single edge between two nodes, it still has an associated complexity since the BOOKEVENT class is part of an inheritance hierarchy. This complexity is the addition of the complexity of each descendents-path involved. Since BOOKEVENT is the root class for this hierarchy, all descendents-paths are involved in the computation of the complexity. Those descendents-path are all of the same size, and thus the same complexity: $2 \times (2 - 1)$. There are 7 descendents-path, the complexity for the inheritance

1345 hierarchy is thus $7 \times 2 \times (2 - 1) = 14$. This is also the
 1346 complexity for P1.

1347 The path P2 goes through bEvt and bSt that correspond to
 1348 two classes that are root classes of inheritance trees. We
 1349 have just computed the complexity of the tree under bEvt,
 1350 which is 14. The complexity for the second inheritance
 1351 tree is computed in the same way, it is the addition of
 1352 the complexity of each descendents-path involved. There
 1353 are two paths of length 2 and three of length 3, so
 1354 the complexity is: $2 \times (2 - 1) + 2 \times 2 - 1 + 3 \times (3 - 1) +$
 1355 $3 \times (3 - 1) + 3 \times (3 - 1) = 22$. The complexity of P2 is the
 1356 product of the two complexities, this corresponds too the
 1357 fact that classes in one inheritance tree can potentially
 1358 interact with every class in the other tree. The path P2 has a
 1359 complexity of $22 \times 14 = 308$.

1360 The total complexity for the class interaction is equal to
 1361 the product of the complexities of P1 and P2. This is the
 1362 maximum number of class-to-class interactions. Of course a
 1363 very large number of them is infeasible (e.g. the setdamaged
 1364 event never interacts with any state) and interactions
 1365 between several classes can be covered by a single test
 1366 case (e.g. TC2 in Section 3.4 covers interactions between 4
 1367 classes). The complexity of an interaction is thus an upper
 1368 bound for the number of relationships that should be
 1369 covered, taking into account all the dependencies in the
 1370 same way. As we see in Section 5, defining roles for
 1371 the relationships would enable to ignore some edges in the
 1372 computation of the complexity, and thus have a value closer
 1373 to the actual number of class-to-class interactions.

1374
 1375

1376 **5. Improving design testability**

1377

1378 Improving testability of the software, with respect to our
 1379 testing criterion, means either avoiding object interactions
 1380 and especially concurrent accesses to shared objects, or
 1381 decreasing the number of potential interactions to have a
 1382 better idea of the actual testability of the design. As we
 1383 suggested in Section 3, a solution may consist in clarifying
 1384 the design, so that the code can be as close as possible to
 1385 what the designer wants.

1386 When it is possible, a way to improve testability and
 1387 break inheritance complexity is to use of interface classes
 1388 that are ‘empty’ from an execution point of view. Never-
 1389 theless it is not possible in all cases. Besides, the UML
 1390 allows a user to define *stereotypes* to associate a semantic to
 1391 UML elements. We thus define several stereotypes that
 1392 specify the semantic of links involved in testability anti-
 1393 patterns (association, dependency, aggregation, compo-
 1394 sition). Thanks to these additional specifications, the
 1395 programmer should avoid implementing an object inter-
 1396 action. As it will be illustrated in Section 6, a simple set of
 1397 refinement actions may be of great help to improve the
 1398 design, suppress ambiguity and reduce the testing effort.
 1399 The stereotypes introduced here are analogous in some way
 1400 to data flow testing criteria for classical software [11], that

1401 identify ‘definition’ and ‘use’ of variables in a program.
 1402 This classical testing model aims at determining the data
 1403 flow, the ‘life line’ of variables at unit level.

1404 Here are the four stereotypes we propose:
 1405

- «create»: a create stereotype on a link from class A to class 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. 1406-1409
- «use»: a use stereotype on a link from class A to class 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: 1410-1413
 - «use_consult»: is a specialization of «use» stereotype where the called methods do never modify attributes of the objects of type B. 1414-1415
 - «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. 1416-1419

1420 The absence of stereotype on a link is equivalent to a
 1421 combination of «use» and «create».

1422 The stereotypes are taken into account by the graph
 1423 model by associating another value to U labels. This also
 1424 allows a designer to estimate the improvement of the design
 1425 after adding stereotypes. It corresponds to step 2 of the
 1426 methodology proposed in Section 2.1. The use of stereo-
 1427 types modifies the identification of objects interactions w.r.t.
 1428 the following properties.

1429 *Assertion 1—objects interaction:* Let P1 and P2 be two
 1430 paths from class C to class D, defining a class interaction
 1431 between C and D. Let e1 be the entry edge of end(P1), e2 be
 1432 the entry edge of end(P2), an objects interaction exists iff
 1433 e1 and e2 have associated stereotypes «use» or
 1434 «use_def».

1435 *Assertion 2—self-usage object interaction:* Let P be a
 1436 path from class C to itself, defining a self-usage class
 1437 interaction for C. Let e be the entry edge of end(P), a self-
 1438 usage object interaction exists iff:
 1439

- e has either «use» or «use_def» stereotype. 1440-1441

1442 Comment: As a consequence, when encountering an
 1443 anti-pattern, if the corresponding assertion is false, due to
 1444 the specified stereotype, it will never generate interaction
 1445 between objects of the final implementation. A static
 1446 analysis may verify that the implementation is consistent
 1447 with stereotypes. The testing task will not focus on
 1448 exhibiting such interactions nor explaining why such
 1449 interactions cannot be tested (w.r.t. the testing criterion). 1450

1451 Fig. 15 illustrates a class interaction. The paths going
 1452 from class C to D which end with an edge stereotyped «use»
 1453 or «use_def», so they cause a contradictory usage of the
 1454 shared provider D by class C.

1455 Automated verifications may check that the code is in
 1456 conformance with stereotypes constraints. For example, the

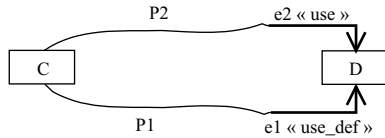


Fig. 15. A class interaction between C and D.

verification of a «use-consult» from A to B consists 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).

Section 6 illustrates potential testability problems on two small architectures, and gives examples of what can be done to avoid real problems at the code level. Stereotypes are introduced directly by the designer, who wants to specify more precisely the software.

6. Application examples

In this section, we apply our testability analysis on two different designs, and for each of them, we propose rules that can improve the testability of these designs. First, we illustrate our approach on the book manager example, then we study a virtual meeting server. The obtained results are useful since they underline the hard points of the designs, where misleading interpretations may occur causing a very

hard to test implementation. The testability analysis for two other case studies is presented in Appendix A.

6.1. The book manager

The CDG for the book manager sub-system is presented Fig. 14. In Section 4.4, we computed the complexity for the class interaction between Bookevent and Book. We mentioned at this moment that many interactions are infeasible, and should thus not be taken into account for the computation of the complexity. In the Section 6 we presented stereotypes that aim at clarifying the model by allocating roles to the relationship. In that way, the different types of relationships could taken into account in different ways when computing the complexity.

The class interaction CI(bEvt, book) can be removed by specifying that the «uses» dependency between the classes BOOKEVENT and BOOK is only for reading. The dependency can be stereotyped «uses_consult», and there is no class interaction anymore. The path going from BOOKEVENT to BOOK through BOOKSTATE is still complex, but could be simplified by refactoring the BOOKEVENT and BOOKSTATE classes into interface classes. This would avoid interactions between those classes and their children, and bring back the complexity of this path to 56 instead of 308.

In the same way, the complexity of the two self-usages interactions SU1(book) and SU2(book) can be reduced by refactoring the BOOKEVENT and BOOKSTATE classes into interface classes.

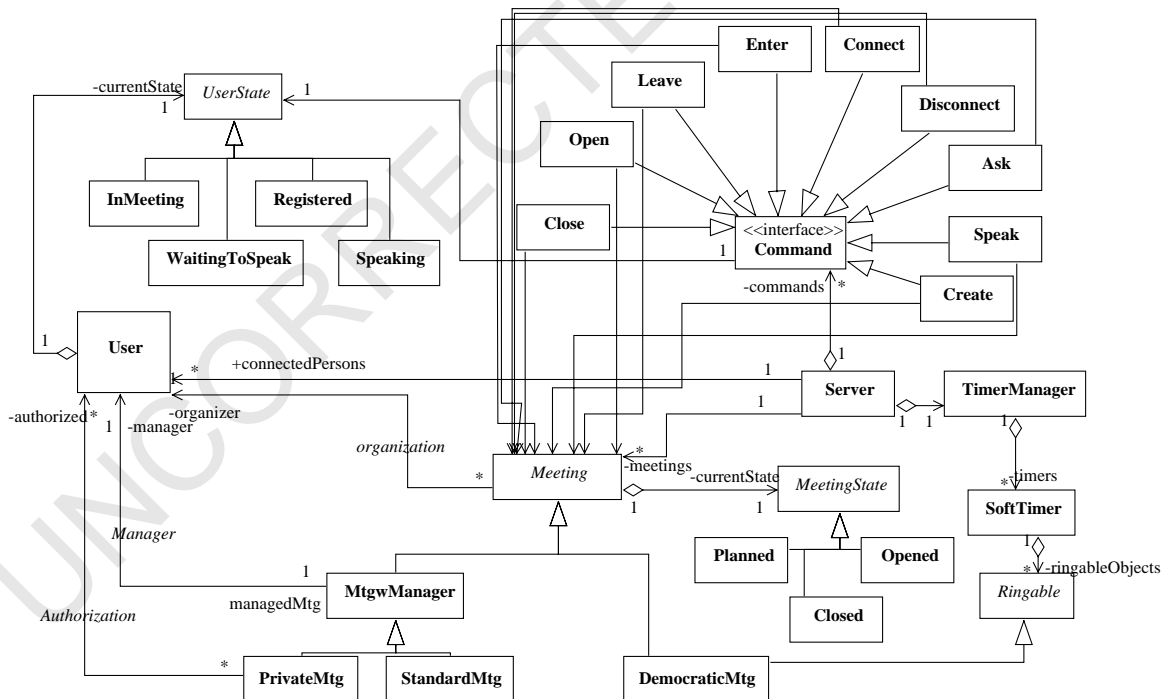


Fig. 16. The Virtual Meeting Server.

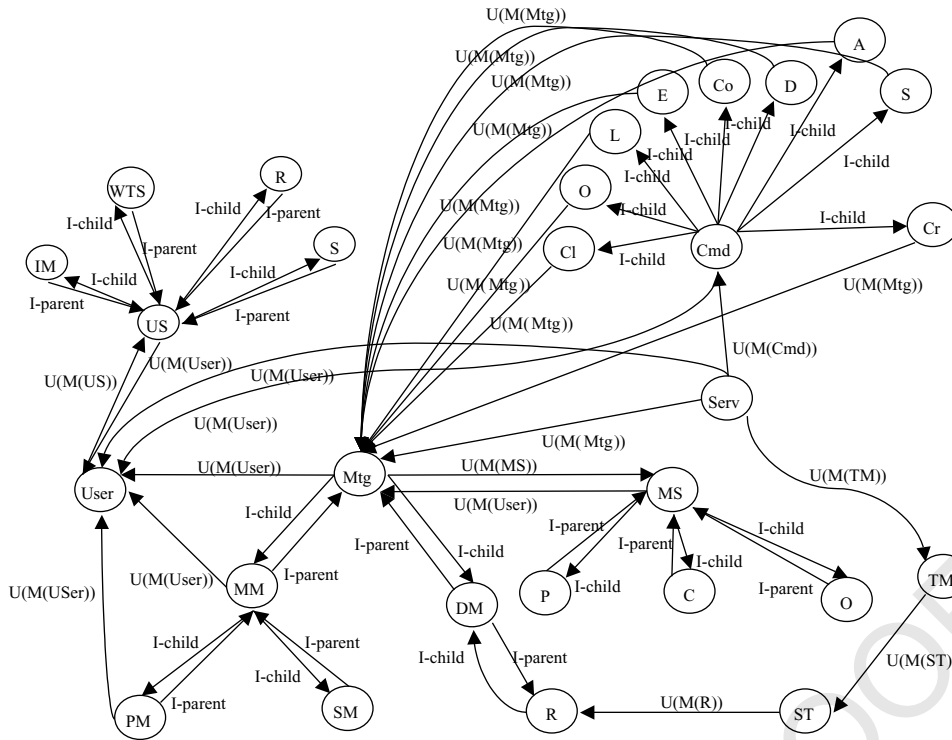


Fig. 17. CDG for the virtual meeting server.

6.2. Testability of a virtual meeting server

Fig. 16 presents the class diagram for a virtual meeting server. This server aims at simulating work meetings. When connected to the server, a client can enter or exit a meeting, speak, or plan new meetings. Three types of meetings exist:

- standards meetings where the client who has the floor is designated by a moderator (nominated by the organizer of the meeting)
- democratic meetings which are standard meetings where the moderator is a FIFO robot (the first client to ask for permission to speak is the first to speak)
- private meetings which are standard meetings with access limited to a defined set of clients.

All the possible commands are reified and inherit of the COMMAND interface. The possible internal states of a client and a meeting are managed through the STATE pattern.

The Class Dependency Graph for the Virtual Meeting Server is given Fig. 17. A lot of class interactions are detected on this model, and we do not detail all of them, but just emphasize interesting configurations, and show that even on a quite simple design (29 classes), a lot of testing problems appear.

There are two self usage interactions around nodes User and Mtg. This is due to the use of a State design pattern [19]. For both of these interactions, it is possible to refactor the USERSTATE and MEETINGSTATE to make interfaces instead of abstract classes. This refactoring does

not suppress the self-usage interactions, but to reduces their complexity.

An interesting configuration of nested class interactions exists between User, Serv, Mtg and MM. There is a class interaction CI(Serv, User) on one hand, and another CI(Mtg, User) on the other hand. Note that CI(Serv, User) includes CI(Mtg, User).

Two remarks can be made on this particular configuration isolated on Fig. 18. First, depending on the way the nested interaction CI(Mtg, User) will be solved, its enclosing class interaction CI(Serv, Mtg) is not necessarily solved. Secondly, even if it is not possible to delete CI(Mtg, User), CI(Serv, User) can be solved, for example refining the design with a stereotype «use_consult» on the association from Server to Meeting. From this configuration, we can deduce that the class interactions can be combined in different ways; in some cases, not all the class interactions have to be taken into account (as in Fig. 11), in others cases, it is necessary to deal with all the class interactions(as in Fig. 18)

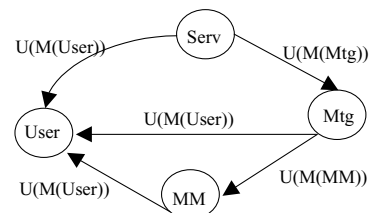


Fig. 18. Configuration of included class interactions.

1681 Others class interactions can be detected, from example
 1682 from Mtg to User (where Mtg can access to User directly,
 1683 through MM or through MM and PM) or from Serv to Mtg.
 1684

1685

1686

1687 7. Related work

1688

1689 Testability is at the border of two software research
 1690 fields. On one hand it is related to testing problems: it
 1691 evaluates the effort needed to test a piece of software. On the
 1692 other hand, the testability is a measurement, thus a large part
 1693 of this work is related to previous work about object-
 1694 oriented metrics.

1694 Traditionally, testing is often divided into several phases,
 1695 for example, unit testing, integration testing and system
 1696 testing. This separation is not so clear for testing of an OO
 1697 system. Due to inheritance and dynamic binding, the control
 1698 flow of an OO-system is not rooted anymore in the main
 1699 encapsulation unit, the class. Unit testing, which focuses on
 1700 classes and methods, cannot capture the interactions
 1701 distributed throughout the system. The effectiveness of
 1702 unit testing is thus even more limited to local aspects [21,22]
 1703 than it is in ‘traditional’ (non-OO) systems. Integration
 1704 testing, on the other hand, insists more on the component
 1705 interfaces and on the order in which components are
 1706 integrated [23–26]. It does not concentrate on testing of
 1707 internal component interaction. Hence, it also may miss
 1708 some of the interactions among the classes. Finally, at the
 1709 system level, testing is usually of the ‘black-box’ nature,
 1710 and is often not formalized, and when it is, it requires, to be
 1711 really applicable in practice, strong (and possibly unrealistic)
 1712 assumptions concerning the completeness of behavioral
 1713 and dynamic models [27]. In this paper, the work we
 1714 propose is complementary to system testing: it aims at
 1715 covering object interdependencies with test cases that may
 1716 be obtained using system testing techniques [28], e.g.
 1717 derived from use cases and sequence/collaboration
 1718 diagrams.

1719 Besides, a large number of measures have been proposed
 1720 to evaluate the quality of object-oriented designs [12], one
 1721 of them is coupling. The coupling measures the strength of
 1722 the relationship between two modules. In the case of object-
 1723 oriented designs, modules are classes. Since the introduc-
 1724 tion of this measure, a large number of coupling measures
 1725 have been proposed, which correspond to different types of
 1726 relationships between classes [29].

1727 This paper proposes a mapping of a coupling
 1728 measurement to precise modeling elements of the UML.
 1729 The coupling between object (CBO) measure [29,30]
 1730 corresponds to a set of classes that use each other’s. In the
 1731 UML class diagram, a class A is said to use another class
 1732 B if there exists an association or a dependency between
 1733 these classes. The CBO measure is discussed in terms of
 1734 testability in [2], and test criteria for this type of
 1735 relationship among classes are proposed in [31]. These
 1736 work focus on each path independently and aims 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.

8. Conclusion

In this paper, we have identified two configurations in
 a UML class diagram that can lead to code difficult to
 test. These configurations are called testability anti-
 patterns, and can be of two types, either class interaction
 or self-usage interaction. Those anti-patterns between
 classes may be implemented as interactions between
 objects in which case, the final software may be very
 difficult to test. The paper proposes a test criterion that
 forces to cover all object interactions. It also defines a
 model that can be derived from a class diagram, and from
 which it is possible to detect, in an unambiguous way all
 the anti-patterns. From this model, it is also possible to
 compute the complexity of anti-patterns which is the
 maximum number of object interactions that could exist
 (and should be tested). The testability measurement
 corresponds to the number and complexity of the anti-
 patterns.

Since this measurement is done from a class diagram, all
 we have are potential interactions that may become real. So
 the complexity is really an estimation of the worst case that
 could appear, and is often much greater than the actual
 complexity of the implementation. A refinement in the
 design could consist in précising the role of the relationships
 between classes, so that the information available at a
 design phase is closer to the implementation. In that case,
 the obtained complexity would be closer to the actual
 complexity of the software. To do so, we propose a set of
 refinement actions based on refactoring and UML
 stereotypes.

A further step in that direction would be the study of
 design patterns [19] as microarchitectures in which the roles
 of associations and dependencies are well-known. The idea
 would be to automatically add stereotypes when applying a
 design pattern on a class diagram.

Appendix A. A compiler architecture and an ICQ client

Fig. A1 gives an object-oriented architecture for a
 compiler taken from [32]. 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.

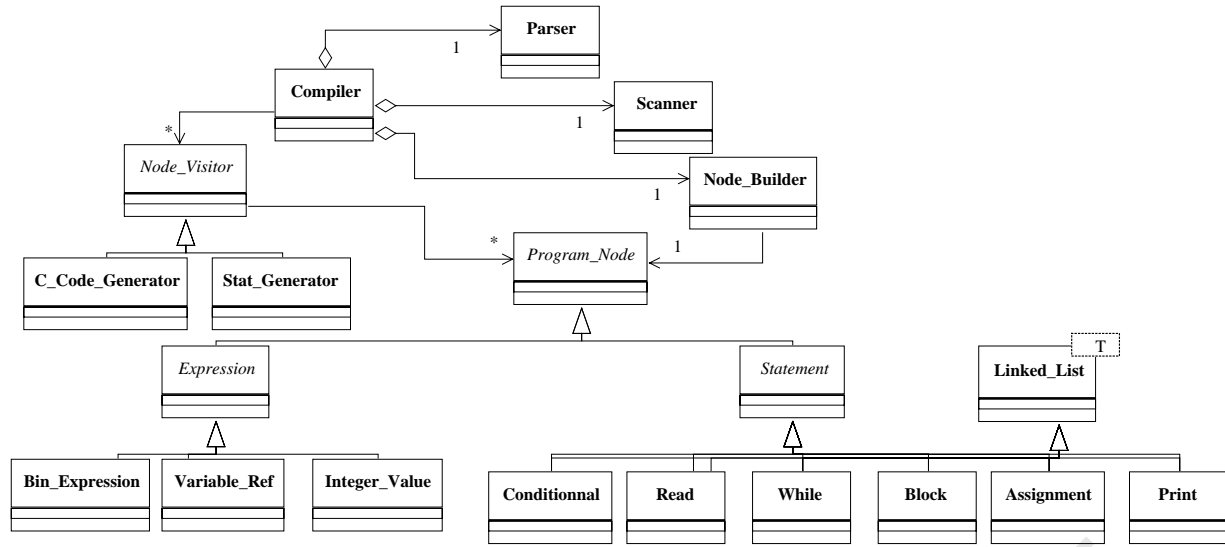


Fig. A1. A compiler architecture.

A Class Dependency Graph can be derived from this architecture (Fig. A2). 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

$$\sum_{i=1}^9 (3(3 - 1)) = 54.$$

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: $54 \times 4 = 216$. In the same way, both interactions have the same complexity that is the product of the two path's complexity: $216 \times 216 = 46656$.

Here, the design can be refined with stereotypes on associations from COMPILER to NODE_VISISTOR and from COMPILER to NODE_BUILDER. Indeed, COMPILER instances should use NODE_VISITOR instances only for queries, the association is thus stereotyped «use_consult». The association from COMPILER to NODE_NUILDER should be stereotyped «use_def» since COMPILER instances might change

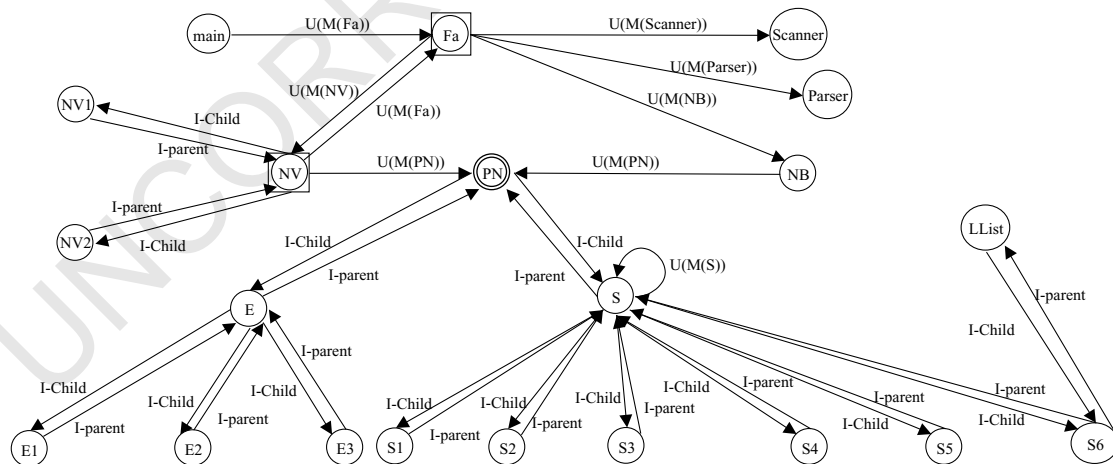


Fig. A2. CDG for the compiler architecture.

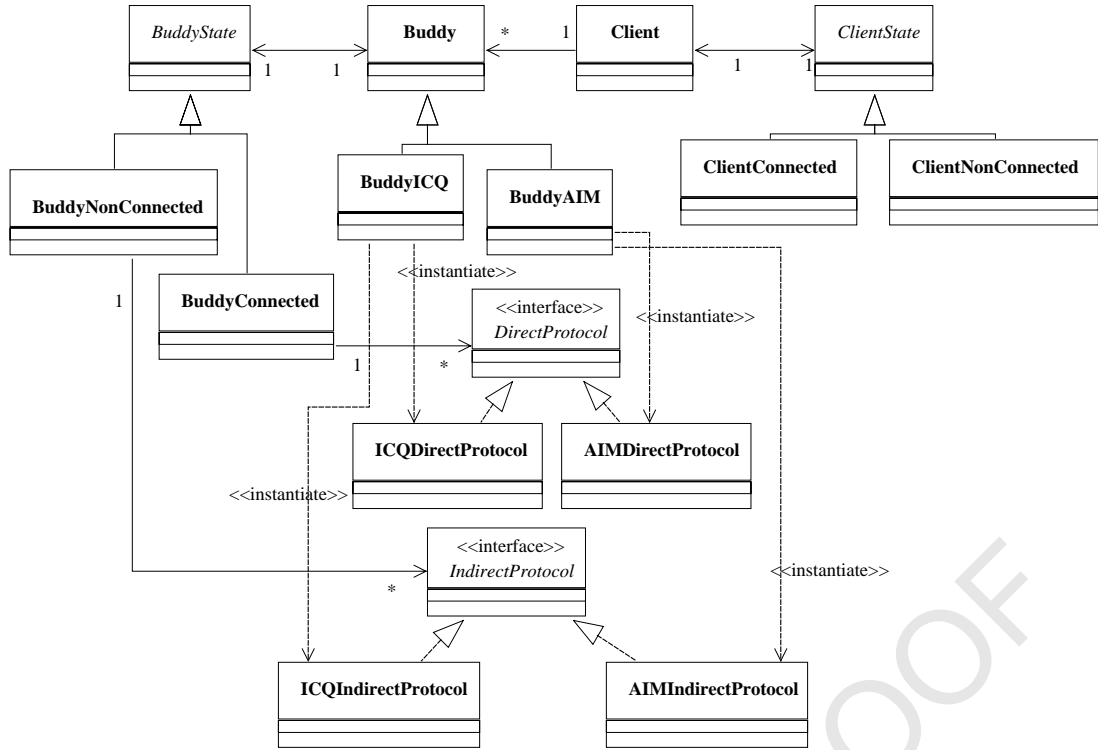


Fig. A3. An instant-messaging client.

the state of NODE_BUILDER instances. If these stereotypes are added to the design, the programmer should not implement any object interactions.

Fig. A3 presents the class diagram for a software that 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.

Several anti-patterns can be detected from the CDG for this system (Fig. A4). Two self-usage interactions SU(c) and SU(b), and four class interactions CI(b,idp), CI(b,adp), CI(b,iip), CI(b,aip). Both self-usage interactions are of complexity 4, and all the class interactions are of complexity 16.

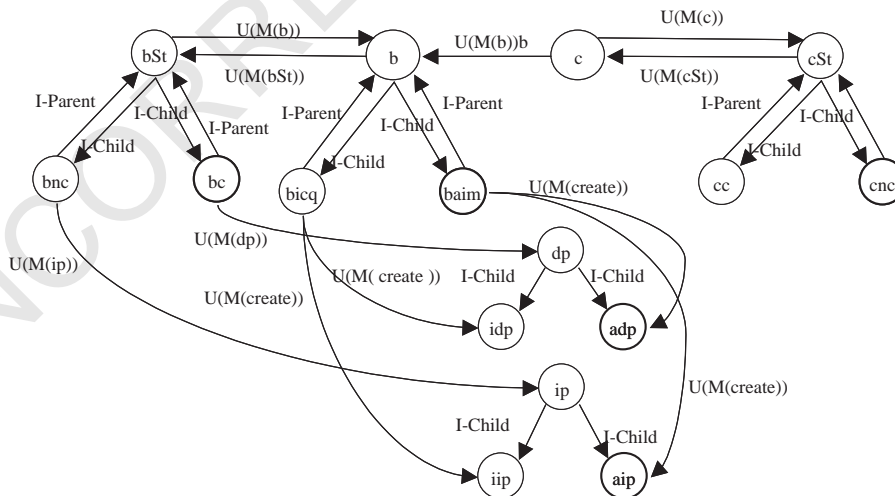


Fig. A4. CDG for an instant-messaging client.

- 2241 [4] R.S. Freedman, Testability of software components, *IEEE Transactions on Software Engineering* 17 (6) (1991) 553–564.
- 2242 [5] J.M. Voas, PIE: a dynamic failure-based technique, *IEEE Transactions on Software Engineering* 18 (8) (1992) 717–727.
- 2243 [6] J.M. Voas, K. Miller, Semantic metrics for software testability, *Journal of Systems and Software* 20 (3) (1993) 207–216.
- 2244 [7] Y. Le Traon, C. Robach. Testability measurements for data flow designs. In: *Proceedings of International Software Metrics Symposium (Metrics'97)*. Albuquerque, NM, USA, November 1997. pp. 91–98.
- 2245 [8] Y. Le Traon, F. Ouabdessalam, C. Robach. Analyzing testability on data flow designs. In: *Proceedings of ISSRE'00 (Int. Symposium on Software Reliability Engineering)*. San Jose, CA, USA, October 2000. pp. 162–173.
- 2246 [9] B.W. Weide, S.H. Edwards, W.D. Heym, T.J. Long, W.F. Ogden. Characterizing observability and controllability of software components. In: *proceedings of 4th International Conference on Software Reuse*. Orlando, USA, April 1996. pp. 62–71.
- 2247 [10] J.M. Bieman, J. Schultz. Estimating the number of test cases required to satisfy the all-du-paths testing criterion. In: *Proceedings of Software Testing Analysis and Verification Symposium*, December 1989. pp. 179–186.
- 2248 [11] S. Rapps, E.J. Weyuker, Selecting software test data using data flow information, *IEEE Transactions on Software Engineering* 11 (4) (1985) 367–375.
- 2249 [12] M. Shepperd, Object-oriented metrics: an annotated bibliography. <http://dec.bournemouth.ac.uk/ESERG/bibliography.html>.
- 2250 [13] N.E. Fenton, R.W. Whitty, Axiomatic approach to software metrication through program decomposition, *Computer Journal* 29 (4) (1986) 330–339.
- 2251 [14] M. Shepperd, D. Ince, *Derivation and Validation of Software Metrics*, Oxford University Press, New York, NY, 1993. p. 167.
- 2252 [15] L. Briand, S. Morasca, V.S. Basili, Property-based software engineering measurement, *IEEE Transactions on Software Engineering* 22 (1) (1996) 68–86.
- 2253 [16] B. Kitchenham, S.L. Pfleeger, N. Fenton, Towards a framework for software measurement validation, *IEEE Transactions on Software Engineering* 21 (12) (1995) 929–944.
- 2254 [17] B. Baudry, J.-M. Jézéquel, Y. Le Traon. Robustness and diagnosability of designed by contracts OO systems. In: *Proceedings of Metrics'01 (Software Metrics Symposium)*. London, UK, April 2001. pp. 272–283.
- 2255 [18] M. Fowler, Reducing coupling, *IEEE Software* 18 (4) (2001) 102–104.
- 2256 [19] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing, Addison-Wesley, 1995.
- 2257 [20] A. Correa, C.M.L. Werner, G. Zaverucha. Object oriented design expertise reuse: an approach based on heuristics, design patterns and anti-patterns. In: *Proceedings of International Conference on Software Reuse*, June 2000. pp. 336–352.
- 2258 [21] J.-M. Jézéquel, D. Deveaux, Y. Le Traon, Reliable objects: a lightweight approach applied to Java, *IEEE Software* 18 (4) (2001) 76–83.
- 2259 [22] M.J. Harrold, G. Rothermel. Performing data flow testing on classes. In: *Proceedings of FSE (Foundation on Software Engineering)*. New Orleans, US, December 1994. pp. 154–163.
- 2260 [23] K. Akif, H. Vu Le, Y. Le Traon, J.-M. Jézéquel. Selecting an efficient OO integration testing strategy: an experimental comparison of actual strategies. In: *Proceedings of ECOOP'01 (European Conference for Object-Oriented Programming)*. Budapest, Hungary, June 2001. pp. 381–401.
- 2261 [24] D.C. Kung, J. Gao, P. Hsia, Y. Toyashima, C. Chen, On regression testing of object-oriented programs, *The Journal of Systems and Software* 32 (1) (1996) 21–40.
- 2262 [25] L. Briand, Y. Labiche. Revisiting strategies for ordering class integration testing in the presence of dependency cycles. In: *Proceedings of ISSRE'01 (Int. Symposium on Software Reliability Engineering)*. Hong-Kong, China, December 2001. pp. 287–296.
- 2263 [26] Y. Le Traon, T. Jéron, J.-M. Jézéquel, P. Morel, Efficient OO integration and regression testing, *IEEE Transactions on Reliability* 49 (1) (2000) 12–25.
- 2264 [27] R.V. Binder, *Testing Object-Oriented Systems: Models, Patterns and Tools*, Addison-Wesley, 1999.
- 2265 [28] L. Briand, Y. Labiche, A UML-based approach to system testing, *Journal of Software and Systems Modeling* 1 (1) (2002) 10–42.
- 2266 [29] L. Briand, J.W. Daly, J.K. Wüst, A unified framework for coupling measurement in object-oriented systems, *IEEE Transactions on Software Engineering* 25 (1) (1999) 91–121.
- 2267 [30] S.R. Shyam, C.F. Kemerer, A metrics suite for object oriented design, *IEEE Transactions on Software Engineering* 20 (6) (1994) 476–493.
- 2268 [31] R.T. Alexander, J. Offutt. Criteria for testing polymorphic relationships. In: *Proceedings of ISSRE'00 (Int. Symposium on Software Reliability Engineering)*. San Jose, CA, USA, October 2000. pp. 15–23.
- 2269 [32] J.-M. Jézéquel, M. Train, C. Mingins, *Design Patterns and Contracts*, Addison-Wesley, 1999. p 348.
- 2270 2297
- 2271 2298
- 2272 2299
- 2273 2300
- 2274 2301
- 2275 2302
- 2276 2303
- 2277 2304
- 2278 2305
- 2279 2306
- 2280 2307
- 2281 2308
- 2282 2309
- 2283 2310
- 2284 2311
- 2285 2312
- 2286 2313
- 2287 2314
- 2288 2315
- 2289 2316
- 2290 2317
- 2291 2318
- 2292 2319
- 2293 2320
- 2294 2321
- 2295 2322
- 2296 2323
- 2297 2324
- 2298 2325
- 2299 2326
- 2300 2327
- 2301 2328
- 2302 2329
- 2303 2330
- 2304 2331
- 2305 2332
- 2306 2333
- 2307 2334
- 2308 2335
- 2309 2336
- 2310 2337
- 2311 2338
- 2312 2339
- 2313 2340
- 2314 2341
- 2315 2342
- 2316 2343
- 2317 2344
- 2318 2345
- 2319 2346
- 2320 2347
- 2321 2348
- 2322 2349
- 2323 2350
- 2324 2351
- 2325 2352