Available online at www.sciencedirect.com
SCIENCE DIRECT
Information and Software Technology xx (xxxx) 1–21
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

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.

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).

0950-5849/$ - see front matter © 2005 Published by Elsevier B.V.
doi:10.1016/j.infsof.2005.01.006

the system. 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 number of object interactions is estimated by the number of 'class interactions': a class interaction is a topological configuration that occurs if a class is supplier from another through various possible paths of dependencies. Moreover, this work proposes a complexity measurement for class interactions based on the complexity of inheritance hierarchies that are present along in the paths of dependencies involved in the interaction. The number of class interactions being an estimate of object interactions (it is actually the maximum number of possible object interactions), it is also an estimate of the number of test cases that will have to be exhibited to test the system. Moreover, the complexity of class interactions is an estimate of the complexity of producing the test cases. The number and complexity of class interactions is thus an estimate of the difficulty of testing a system, and it is the testability measurement proposed in this paper.

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

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,
- to identify hard-to test interactions and measure their number and complexity due to polymorphic uses, considered here as our estimate of design testability,
- to suggest improvements on the design to reduce the number and complexity of class interactions: these improvements at design level are realistic since static verifications on the code ensure their implementation,
- *in fine* to 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.

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

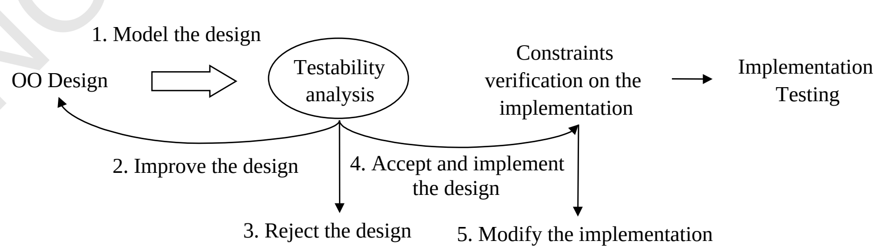


Fig. 1. Improving testability of OO designs.

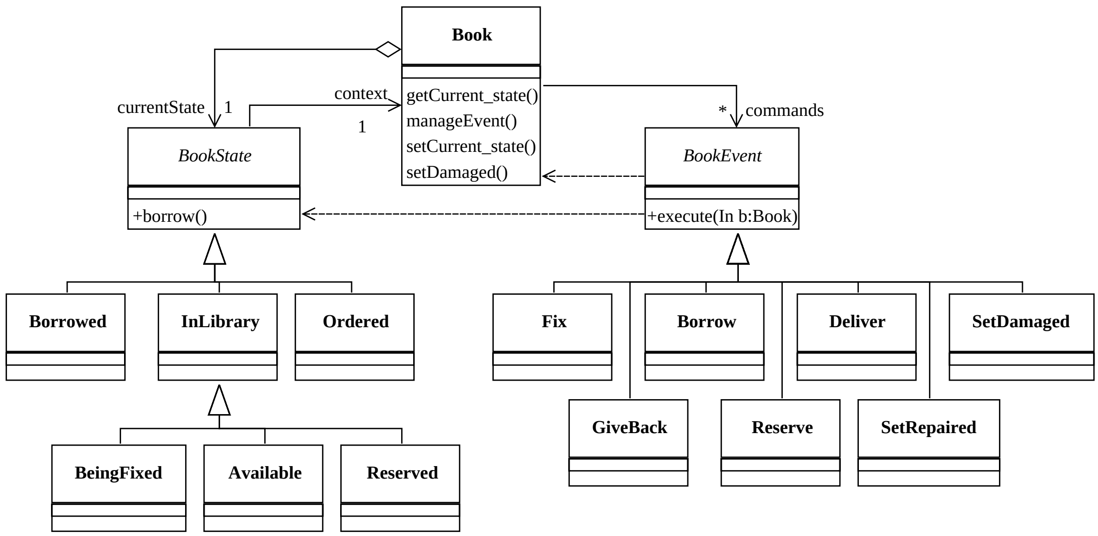


Fig. 2. UML class diagram for book manager sub-system.

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 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

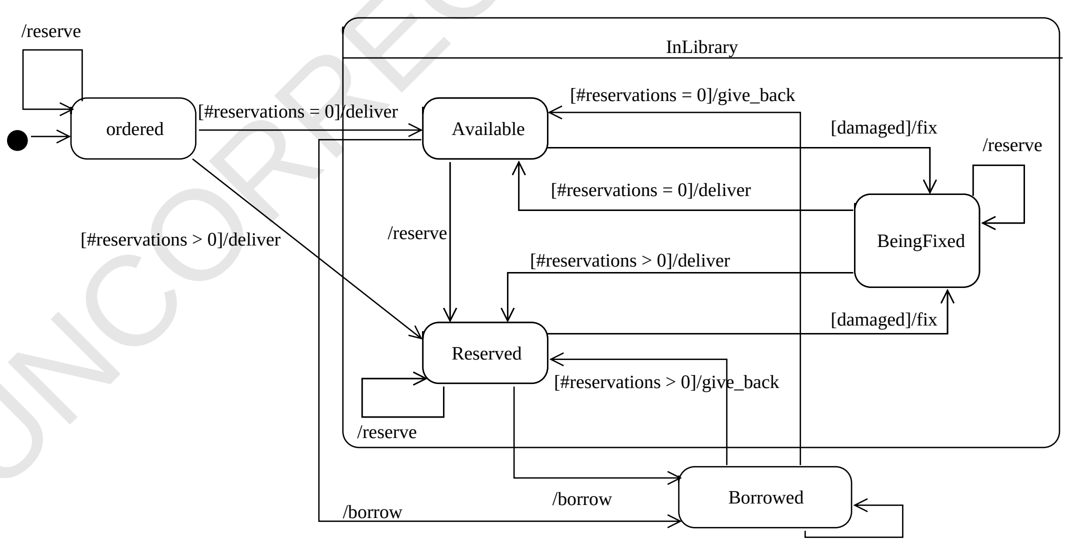


Fig. 3. UML statechart for book management.

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 results 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.

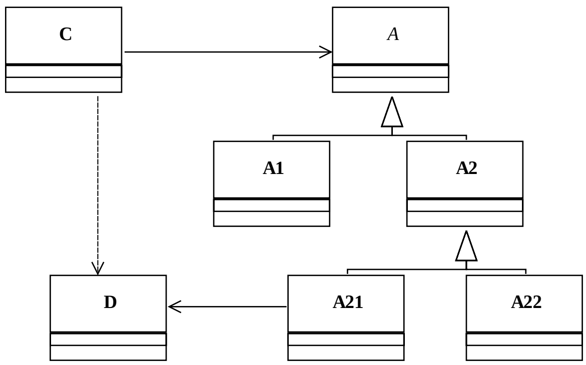


Fig. 4. Concurrent usage through an inheritance hierarchy.

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

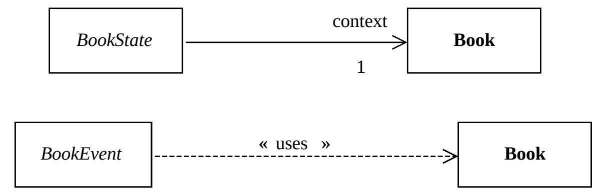


Fig. 5. Association or dependency between classes.

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

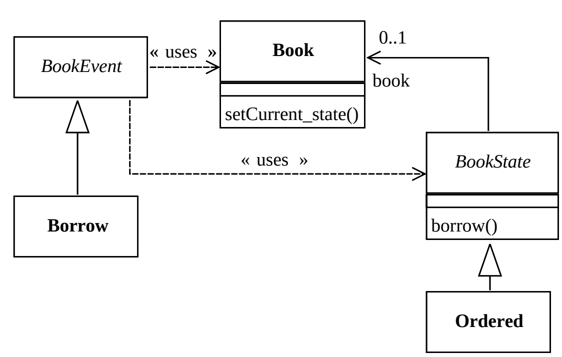


Fig. 6. Transitive relationship between BookEvent and Book through BookState.

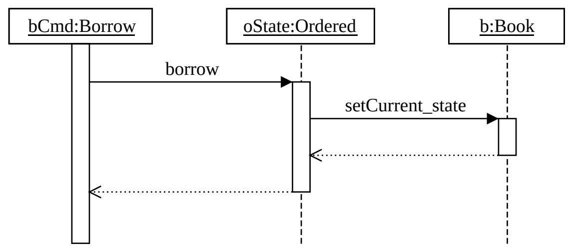


Fig. 7. Sequence diagram illustrating a real indirect relationship between BookEvent and Book.

between the two classes specifies a direct usage relationship. The second relationship is an transitive one through the BookState class. The BookEvent class depends on the BookState class which depends on Book. Thus BookEvent may depend transitevily 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 are not implemented as object interactions, a report stating this absence of actual interaction has to be produced.

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

3.4.1. SU1 interaction

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

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

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

```
public void testManageEvent(){
  Book b=new Book();
  b.manageEvent("setDamaged");
}
```

3.4.2. SU2 interaction

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

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

3.4.3. CI interaction

The third objective is to exhibit an object interaction between BookEvent and Book through two different paths (CI). Since the Book class is the entry point for testing, the test case has to call a method that uses the `commands` set. When writing a test case for the first test objective, we have seen that a call to the `manageEvent()` method covers the relationship from Book to BookEvent, and also the one from BookEvent to Book. Thus the direct path from BookEvent to Book is covered by test case calling `manageEvent()` in the Book class. To cover the second path from BookEvent to Book (through BookState), the call to `manageEvent()` has to cover the relationship between BookEvent and BookState. This can be done by calling an event which processing depends on the actual state of the Book instance. In that case the `execute()` method in the concrete events has the following form:

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

Then, if a transition in the statechart is triggered by the called event, then the relationship between BookState and Book is covered, since in that case the method in the concrete state calls a method on the context attribute. For example, the `borrow()` method in the Available class has to change the state of the context to which it is associated, since the borrow event in the Available state triggers a transition from Available to the Borrowed state. Here is the corresponding code:

```
class Available{
  public void borrow(){context.chan-
  geState(new Borrowed());}
}
```

To summarize this third test objective, the following test case covers the interaction between BookEvent and Book through two different paths and Fig. 8 gives the sequence diagram for this test case (TC2).

```
public void testManageEvent(){
  Book b=new Book(); //the book is in the
  ordered state
  b.manageEvent('deliver'); //puts the
  book in the available state
  b.manageEvent('borrow');
}
```

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

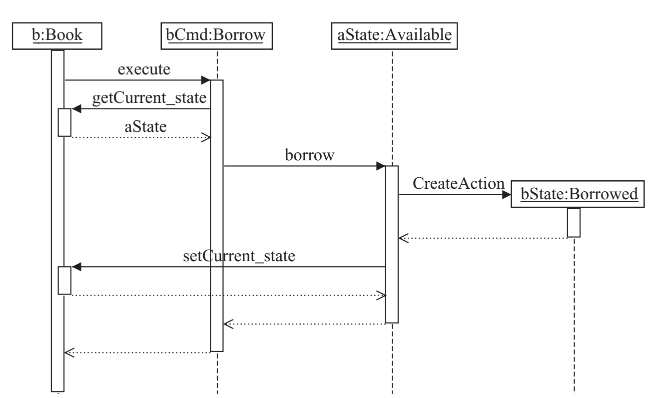


Fig. 8. Sequence diagram for test case n^{o}3.

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.

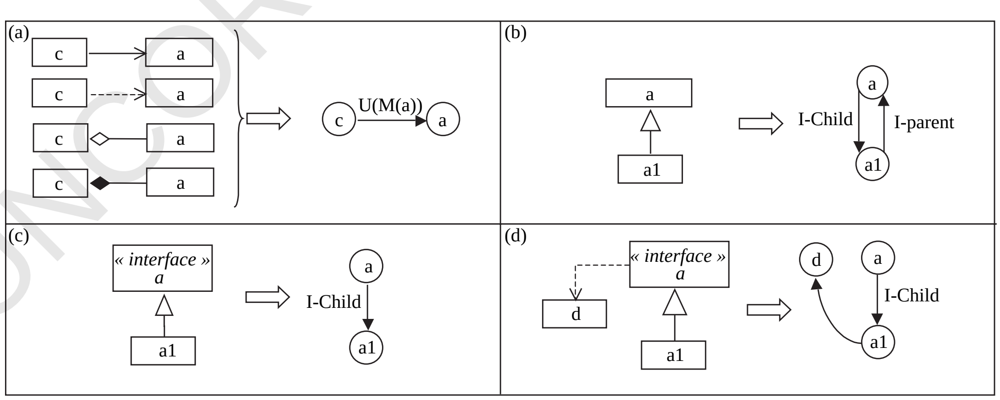


Fig. 9. Basic transformations from a UML class diagram to a CDG.

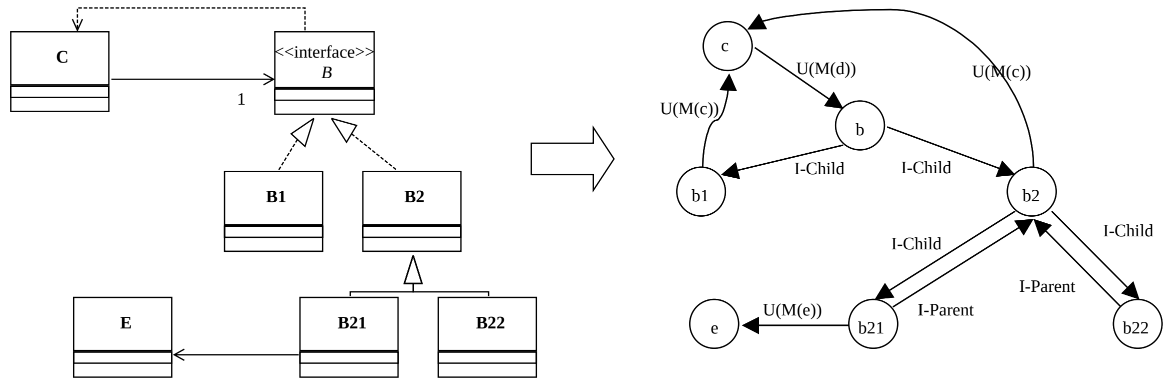


Fig. 10. CDG example.

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 $P = [x_{i1}, x_{i2}, x_{i3}, \ldots, x_{ik}]$, such that:

$(x_{i1}, x_{i2}) \in \Gamma$, $(x_{i2}, x_{i3}) \in \Gamma, \ldots, (x_{ik-1}, x_{ik}) \in \Gamma$
x_{i1} is the origin of the path and is called origin(P)
x_{ik} the end and is called end(P)
the x_{ij} $(2 \le j \le k-1)$, are the intermediate vertices (we call the set of intermediate vertices itVertices(P)).

Definition. *Cycle.* Let P be a path, P is a cycle if and only if end(P) = 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, $P_1 \ne P_2$ 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.

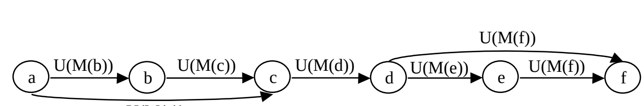


Fig. 11. CI on a CDG.

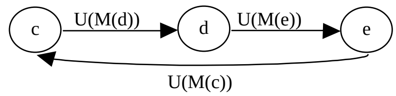


Fig. 12. SU on a CDG.

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 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 $P_1,\ldots,P_{\mathrm{nbPaths}}$ 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:

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

The complexity of a path is defined in the following.

Definition. *Descendents-path*. In an inheritance hierarchy, a *descendents-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 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 $n(n-1)$, because

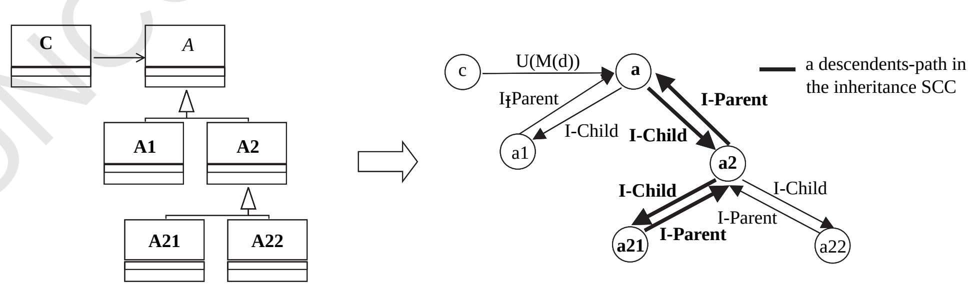


Fig. 13. Slice in a SCC corresponding to an inheritance hierarchy.

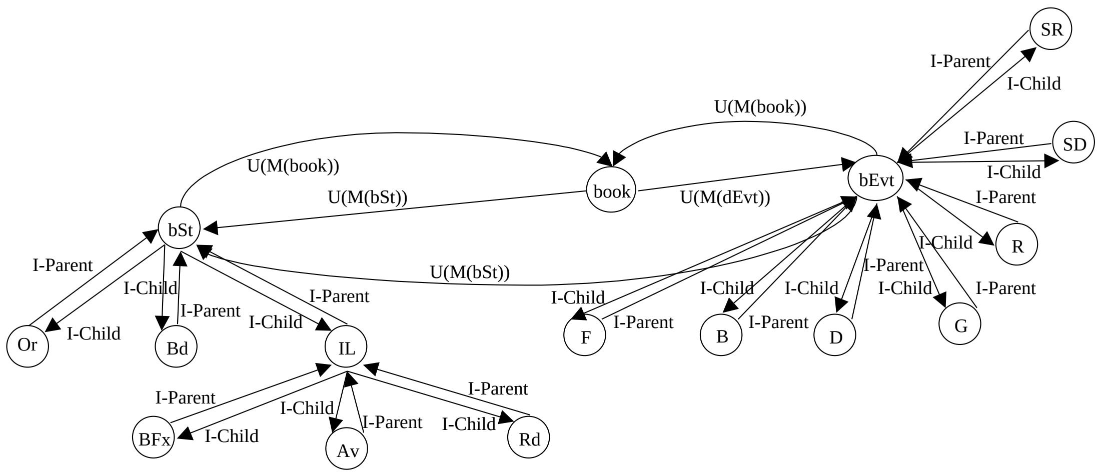


Fig. 14. CDG for the book manager system.

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, $IH_1,\ldots,IH_{\text{nbCrossed}}$ be nbCrossed inheritance hierarchies crossed by P

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

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 $dp_1,\ldots, dp_{\text{nbDP}}$, the nbDP descendents-path in *IH* influencing *P*'s complexity.

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

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:

$$\text{complexity}(dp) = h(h-1)$$

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 hierarchy is thus $7\times2\times(2-1)=14$. This is also the complexity for P1.

The path P2 goes through bEvt and bSt that correspond to two classes that are root classes of inheritance trees. We have just computed the complexity of the tree under bEvt, which is 14. The complexity for the second inheritance tree is computed in the same way, it is the addition of the complexity of each descendents-path involved. There are two paths of length 2 and three of length 3, so the complexity is: $2\times(2-1)+2\times2-1)+3\times(3-1)+3\times(3-1)+3\times(3-1)=22$. The complexity of P2 is the product of the two complexities, this corresponds too the fact that classes in one inheritance tree can potentially interact with every class in the other tree. The path P2 has a complexity of $22\times14=308$.

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

5. Improving design testability

Improving testability of the software, with respect to our testing criterion, means either avoiding object interactions and especially concurrent accesses to shared objects, or decreasing the number of potential interactions to have a better idea of the actual testability of the design. As we suggested in Section 3, 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 to use of interface classes that are 'empty' from an execution point of view. Nevertheless 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 testability anti-patterns (association, dependency, aggregation, composition). Thanks to these additional specifications, the programmer should avoid implementing an object interaction. As it will be illustrated in Section 6, a 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 [11], 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.

Here are the four stereotypes we propose:

- «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.
- «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:
  – «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.

The absence of 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 a designer to estimate the improvement of the design after adding stereotypes. It corresponds to step 2 of the methodology proposed in Section 2.1. The use of stereotypes modifies the identification of objects interactions w.r.t. the following properties.

Assertion 1—objects interaction: Let P1 and P2 be two paths from class C to class D, defining a class interaction between C and D. Let e1 be the entry edge of end(P1), $e2$ be the entry edge of end(P2), an objects interaction exists *iff*

$e1$ and $e2$ have associated stereotypes «use» or «use_def».

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

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

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

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

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

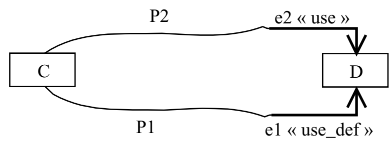


Fig. 15. A class interaction between C and D.

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.

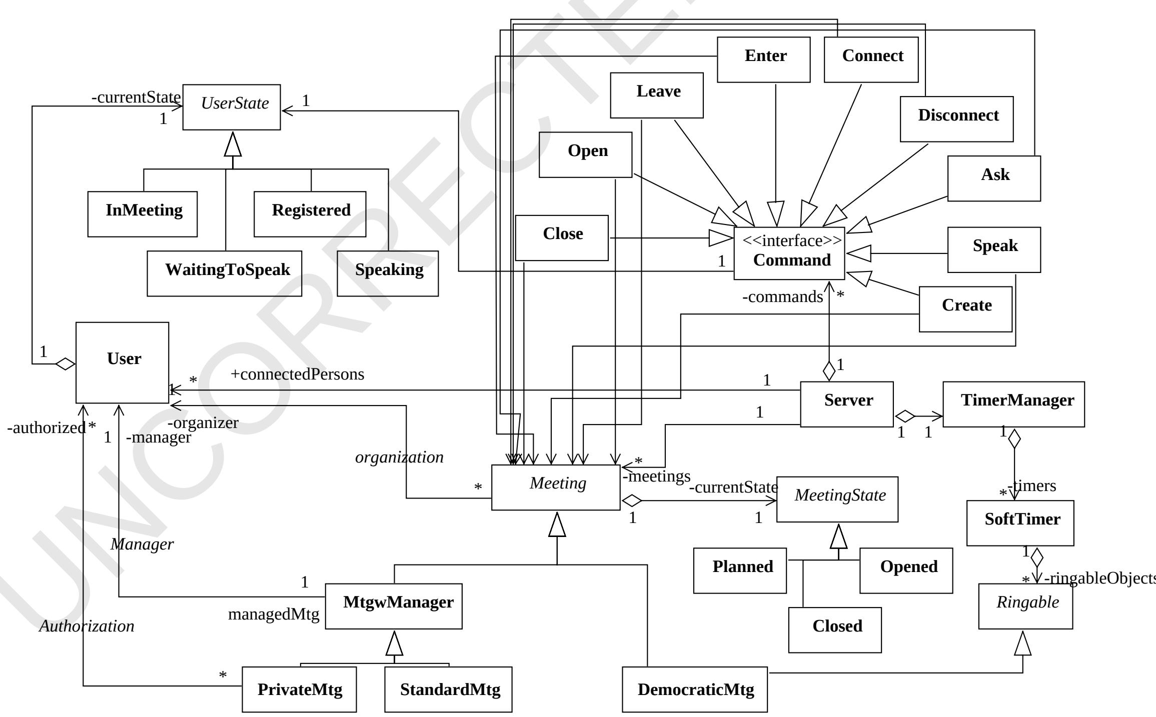


Fig. 16. The Virtual Meeting Server.

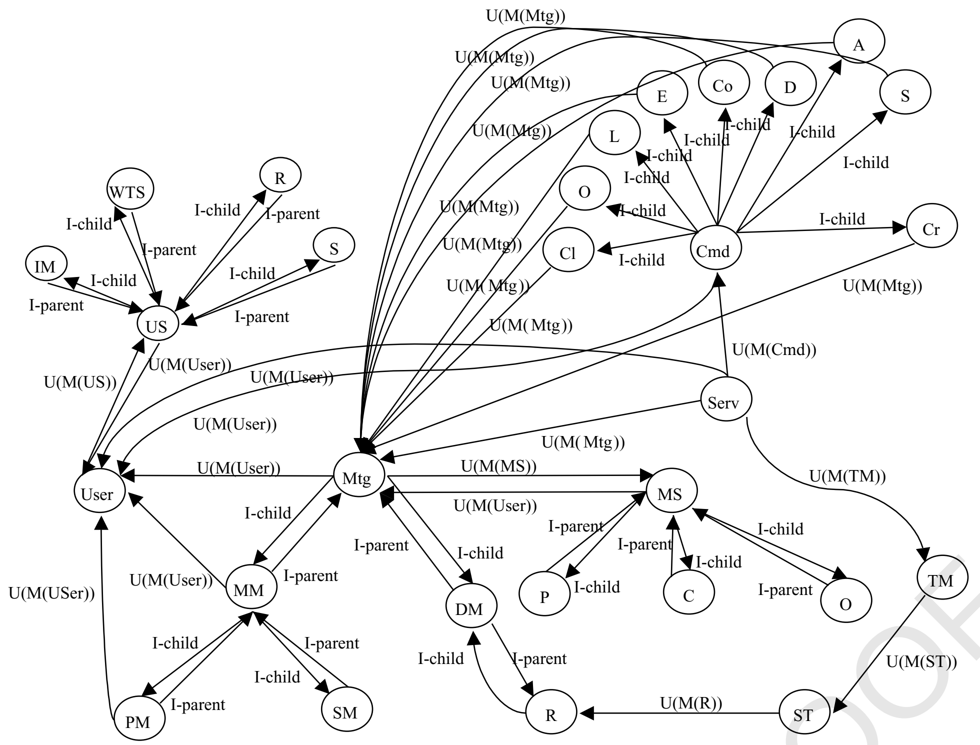


Fig. 17. CDG for the virtual meeting server.

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)

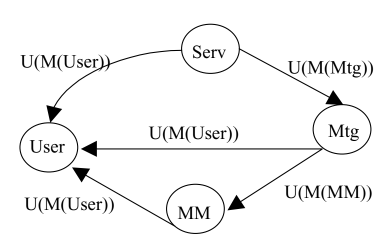


Fig. 18. Configuration of included class interactions.

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

7. 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 often divided into several phases, for example, 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 [21,22] 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 [23–26]. It does not concentrate on testing of internal component interaction. Hence, 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 is often not formalized, and when it is, it requires, to be really applicable in practice, strong (and possibly unrealistic) assumptions concerning the completeness of behavioral and dynamic models [27]. In this paper, the work we propose is complementary to system testing: it aims at covering object interdependencies with test cases that may be obtained using system testing techniques [28], 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 [12], one of them is coupling. The coupling measures the strength of the relationship between two modules. In the case of object-oriented designs, modules are classes. Since the introduction of this measure, a large number of coupling measures have been proposed, which correspond to different types of relationships between classes [29].

This paper proposes a mapping of a coupling measurement to precise modeling elements of the UML. The coupling between object (CBO) measure [29,30] corresponds to a set of classes that use each other's. In the UML class diagram, a class A is said to use another class B if there exists an association or a dependency between these classes. The CBO measure is discussed in terms of testability in [2], and test criteria for this type of relationship among classes are proposed in [31]. These 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.

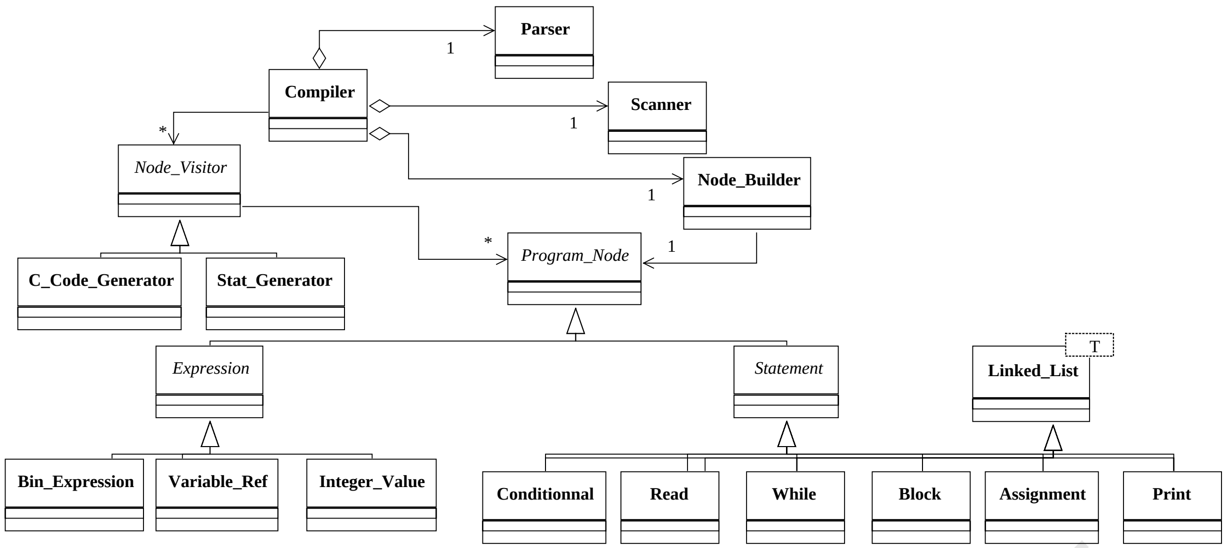


Fig. A1. A compiler architecture.

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: $54\times4=216$. In the same way, both interactions have the same complexity that is the product of the two path's complexity: $216\times216=46656$.

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

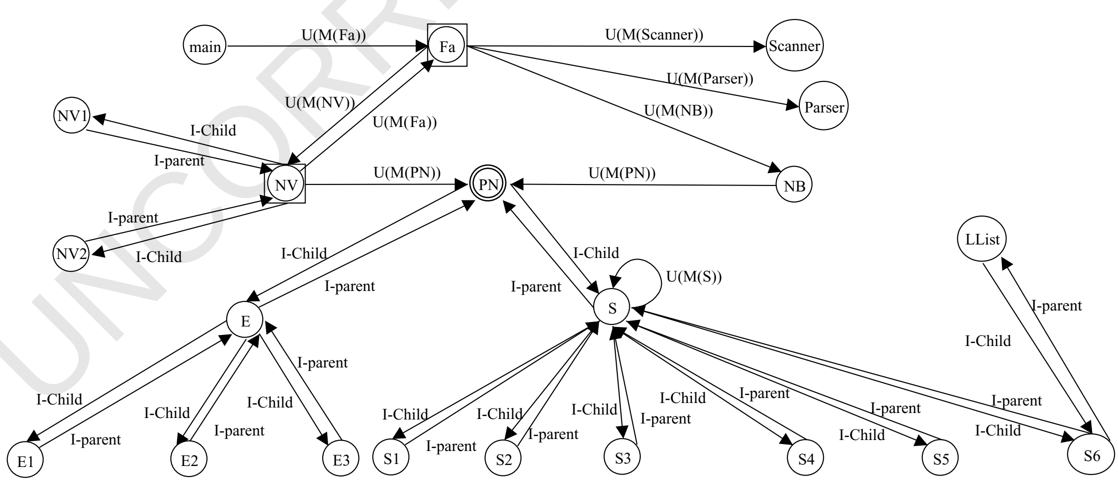


Fig. A2. CDG for the compiler architecture.

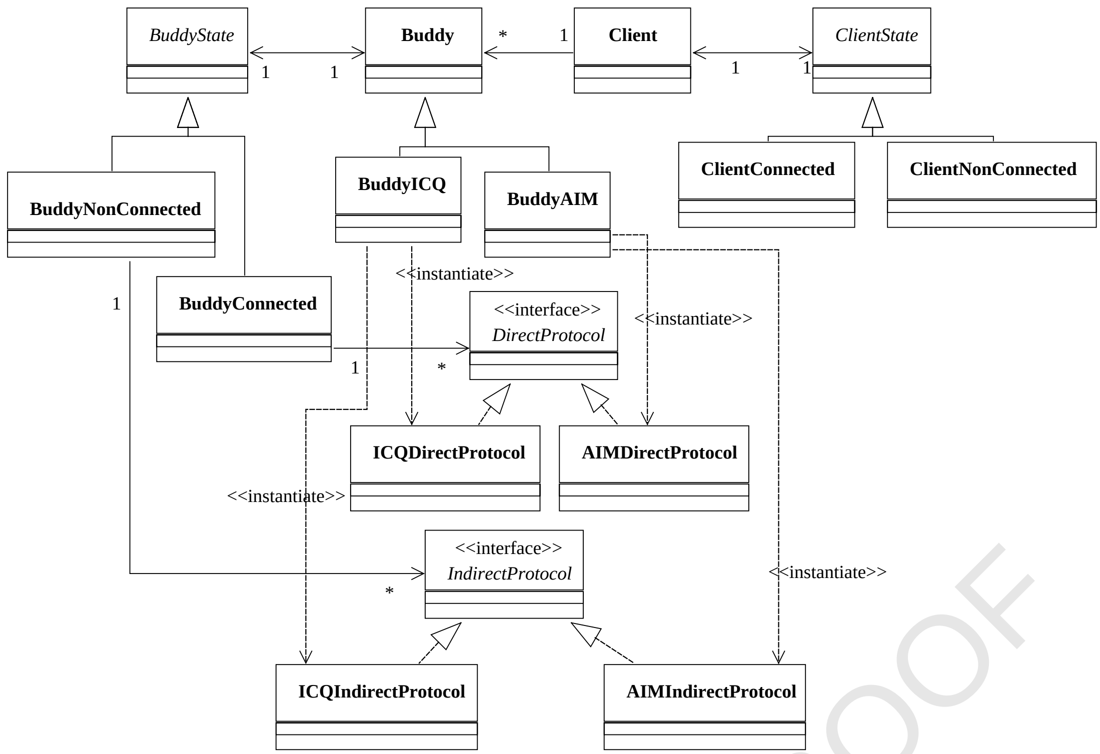


Fig. A3. An instant-messaging client.

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.

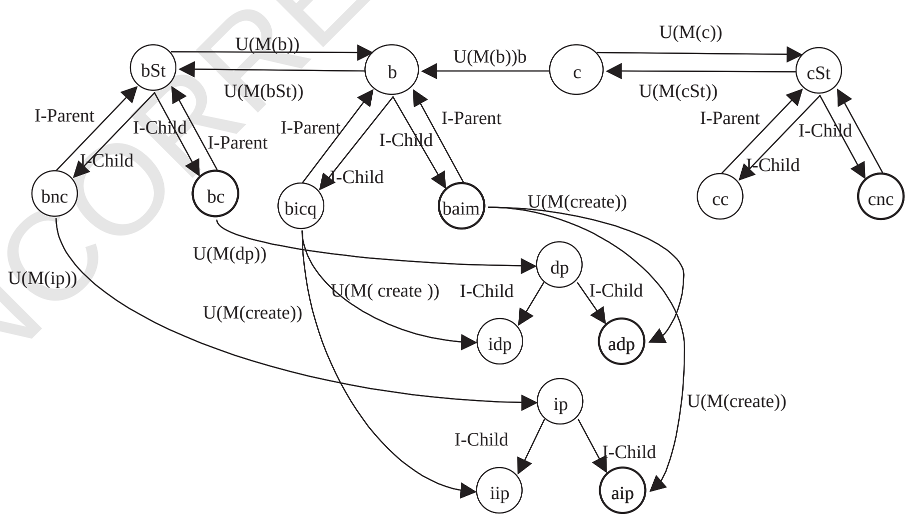


Fig. A4. CDG for an instant-messaging client.

Test case 1

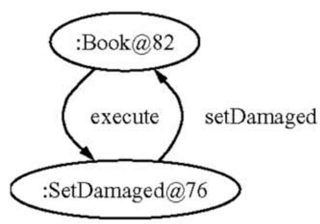
Test data :

```
public void testManageEvent(){
   Book b = new Book();
   b.manageEvent("setDamaged");//the trace is produced from this call
}
```

Textual trace for TC1:

```
|  |  |  |  |  |->SetDamaged@76.execute
|  |  |  |  |  |  |->Book@82.setDamaged
```

Object diagram for TC2:



Test case 2

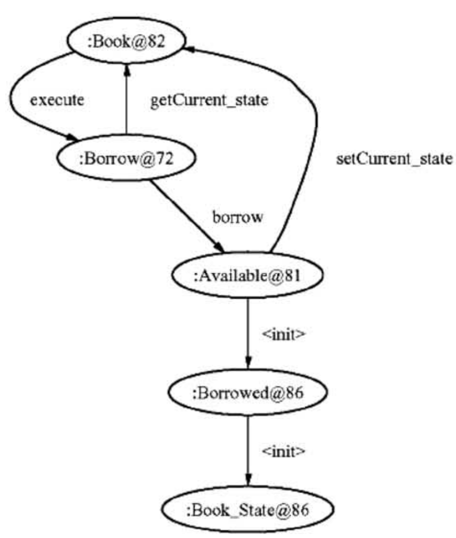
Test data :

```
public void testManageEvent(){
   Book b = new Book(); //the book is in the ordered state
   b.manageEvent("deliver"); //puts the book in the available state
   b.manageEvent("borrow"); //the trace is produced from this call
}
```

Textual trace for TC1:

```
|  |  |  |  |  |->Borrow@72.execute
|  |  |  |  |  |  |->Book@82.getCurrent_state
|  |  |  |  |  |  |->Available@81.borrow
|  |  |  |  |  |  |  |->Borrowed@86.<init>
|  |  |  |  |  |  |  |  |->Book_State@86.<init>
|  |  |  |  |  |  |  |->Book@82.setCurrent_state
```

Object diagram for TC2:



Another more complex test case : TC3 it covers interactions between BOOKEVENT and BOOK. one of the path goes through AVAILABLE, RESERVED and BOOKSTATE

Test case 3

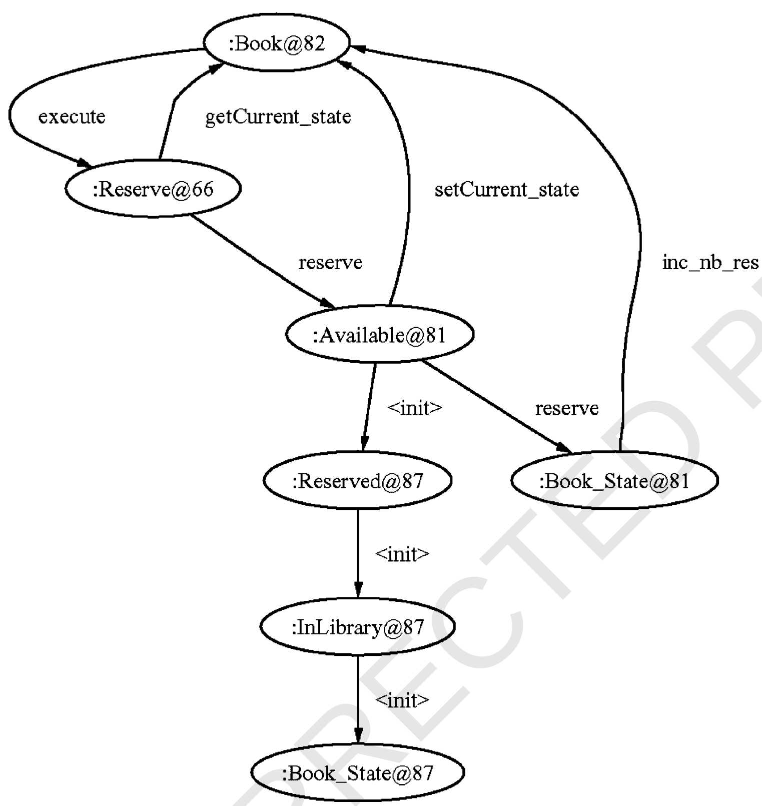
Test data :

```
public void testManageEvent(){
   Book b = new Book(); //the book is in the ordered state
   b.manageEvent("reserve"); // the trace is produced from this call
}
```

Textual trace for TC3:

```
|  |  |  |  |  |->Reserve@66.execute
|  |  |  |  |  |  |->Book@82.getCurrent_state
|  |  |  |  |  |  |->Available@81.reserve
|  |  |  |  |  |  |  |->Reserved@87.<init>
|  |  |  |  |  |  |  |  |->InLibrary@87.<init>
|  |  |  |  |  |  |  |  |  |->Book_State@87.<init>
|  |  |  |  |  |  |  |->Book@82.setCurrent_state
|  |  |  |  |  |  |  |->Book_State@81.reserve
|  |  |  |  |  |  |  |  |->Book@82.inc_nb_res
```

Object diagram for TC3:



Appendix B. J Tracor book manager system study

We present here the main traces and the object graphs obtained with the Jtracor tool available from http://franck.fleurey.free.fr/JTracor/index.htm. JTracor is a framework and has a specific output format for traces that can be easily implemented. Here is the simple kind of results we produce with the standard version, a textual and a graph representing the objects instances and their message exchanges. A future version will produce UML sequence diagram in the XMI standard exchange format of UML. The tool has been successfully applied on very complex system calls involving hundred of communicating objects (such as call on the Java virtual machine).

References

[1] J.M. Voas, K. Miller, Software testability: the new verification, IEEE Software 12 (3) (1995) 17–28.
[2] R.V. Binder, Design for testability in object-oriented systems, Communications of the ACM 37 (9) (1994) 87–101.
[3] A. Abdurazik, A.J. Offutt. Using UML collaboration diagrams for static checking and test generation. In Proceedings of UML'00. York, UK, October 2000. pp. 383–395.

[4] R.S. Freedman, Testability of software components, IEEE Transactions on Software Engineering 17 (6) (1991) 553–564.

[5] J.M. Voas, PIE: a dynamic failure-based technique, IEEE Transactions on Software Engineering 18 (8) (1992) 717–727.

[6] J.M. Voas, K. Miller, Semantic metrics for software testability, Journal of Systems and Software 20 (3) (1993) 207–216.

[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.

[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.

[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.

[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.

[11] S. Rapps, E.J. Weyuker, Selecting software test data using data flow information, IEEE Transactions on Software Engineering 11 (4) (1985) 367–375.

[12] M. Shepperd, Object-oriented metrics: an annotated bibliography. http://dec.bournemouth.ac.uk/ESERG/bibliography.html.

[13] N.E. Fenton, R.W. Whitty, Axiomatic approach to software metrication through program decomposition, Computer Journal 29 (4) (1986) 330–339.

[14] M. Shepperd, D. Ince, Derivation and Validation of Software Metrics, Oxford University Press, New York, NY, 1993. p. 167.

[15] L. Briand, S. Morasca, V.S. Basili, Property-based software engineering measurement, IEEE Transactions on Software Engineering 22 (1) (1996) 68–86.

[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.

[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.

[18] M. Fowler, Reducing coupling, IEEE Software 18 (4) (2001) 102–104.

[19] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing, Addison-Wesley, 1995.

[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.

[21] J.-M. Jézéquel, D. Deveaux, Y. Le Traon, Reliable objects: a lightweight approach applied to Java, IEEE Software 18 (4) (2001) 76–83.

[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.

[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.

[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.

[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.

[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.

[27] R.V. Binder, Testing Object-Oriented Systems: Models, Patterns and Tools, Addison-Wesley, 1999.

[28] L. Briand, Y. Labiche, A UML-based approach to system testing, Journal of Software and Systems Modeling 1 (1) (2002) 10–42.

[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.

[30] S.R. Shyam, C.F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Enginnering 20 (6) (1994) 476–493.

[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.

[32] J.-M. Jézéquel, M. Train, C. Mingins, Design Patterns and Contracts, Addison-Wesley, 1999. p 348.