Improving the Testability of UML Class Diagrams

Benoit Baudry, Yves Le Traon

IRISA, Campus Universitaire de Beaulieu,
35042 Rennes Cedex, France
{Benoit.Baudry, Yves.Le_Traon}@irisa.fr

Gerson Sunyé

LINA, 2 rue de la Houssinière,
44322 Nantes Cedex 03, France
sunye@lina.univ-nantes.fr

Abstract

This paper synthesizes our research efforts in the field of object-oriented test. These efforts have two different goals. First, we identify of recurrent design structures — or testability anti-patterns — that worsen software testability. Second, we use the UML extension mechanisms to better specify design information that can make implementation more testable. Although *detecting testability anti-patterns during software design is a crucial task, one cannot expect from a non-specialist to make the right improvements, without guidance or automation. To overcome this limitation, each definition of an anti-pattern is associated with an alternative design solution.*

1. Introduction

Software testing is often a very costly part of its life cycle. A lot of work focus on reducing the testing cost working on a wide variety of aspects that have an impact on the test activity: automatic generation, test harnesses and frameworks, efficient test planning… Another activity that can help reduce the testing effort consists in taking into account testing during the analysis and design stages, to build software systems that will offer good properties to be tested. This property to be more or less easily tested is called *testability* [1]. The testability is an important criterion for software developers since the sooner it can be estimated, the better the software architecture will be organized to improve subsequent test and maintenance.

Testability has been revived with the object-orientation ([2]), and this work is concerned with the issue of testability of object-oriented (OO) static designs based on the UML (Unified Modeling Language) class diagram. The key idea is to identify parts of the class diagram that can make the testing of the implementation difficult. A testability measurement for class diagrams is thus proposed in that intent. The measurement aims at pinpointing points of the design that can decrease the testability of the implementation. These points are called *testability anti-patterns*. The measurement points these anti-patterns in a class diagram and associates a complexity value. Once these anti-patterns are identified, they can be taken into account by designers to improve the class diagram at these particular points.

A second contribution of the work presented here is to propose solutions to improve the testability of the design. The idea is to add information on the design to make the implementation more testable. UML extension mechanisms such as tagged values and stereotypes are the main feature used to specify the roles of the relationships in the class diagram.

This paper synthesizes several ideas we have investigated in previous work. Our goal here is to give the intuition of the testability issues that can be detected from class diagrams and how these models can be improved to make the design more testable. In the following we thus introduce an example and illustrate our global approach on this example. We intentionally avoid the presentation of the formal aspects of this work, which have been published in [3].

2. Example

We introduce here a UML class diagram that serves as an illustration example all along this paper. This diagram corresponds to a subsystem in charge of managing books in a larger library system (Figure 1). All the classes are given but only the methods that are used to illustrate particular points in the following sections are displayed. This class diagram is the design for a system implementing the UML statechart presented Figure 2. The statechart describes the dynamic behavior of a book object. An object is created 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 (Figure 1) is based on two design patterns [4]: the state pattern that reifies each state of the statechart in one class and the command pattern that reifies events. A possible implementation for this class diagram is available here: http://www.irisa.fr/triskell/results/Book/.

Based on this simple example the rest of the paper identifies two testability anti-patterns and an associated complexity measurement. Then, we propose solutions to improve the design on the particular parts of the design pinpointed by the anti-patterns.
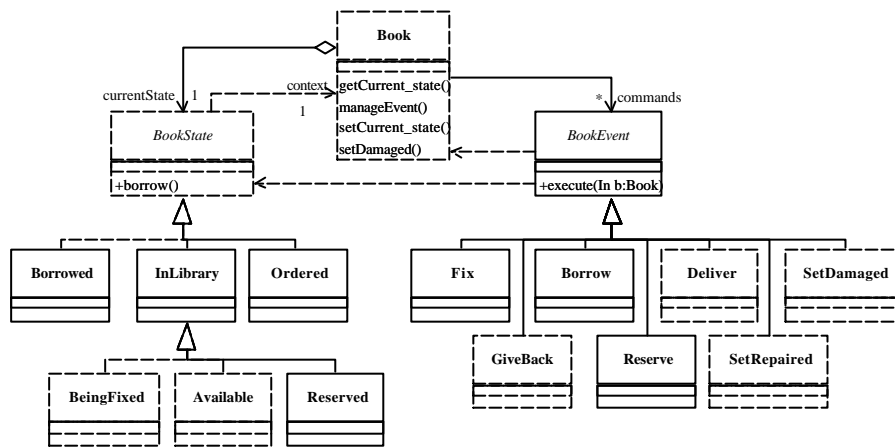

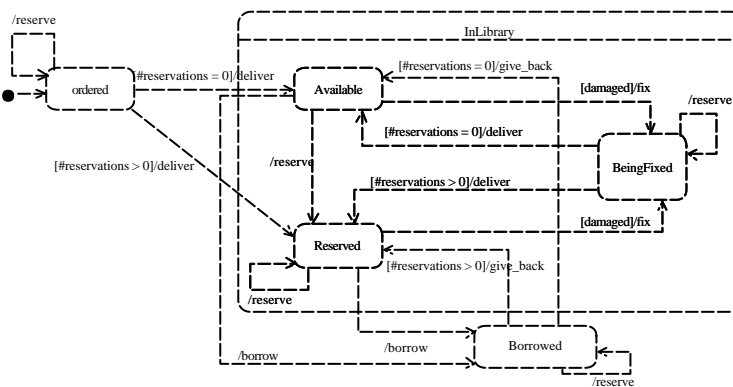**Figure 1 – UML class diagram for book manager sub-system**

Figure 2 – UML statechart for book management

3.   Testability anti-patterns for OO systems

Like for any classical software, the difficulty for testing object-oriented systems is due to the existence of client/supplier relationships in 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 section starts with an informal analysis of testability problems that can be detected from a class diagram, using the book manager design example (Figure 1). 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 test criterion is then the basic reference to compute the testability of the class diagram: the testability is evaluated as the effort needed to generate test cases that satisfy the test criterion. The section ends with examples of test generation for the book manager, using 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 its corresponding implementation. The class diagram given in Figure 1 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. As said in the introduction, these weaknesses are called *testability anti-patterns*. An anti-pattern describes a solution to a recurrent problem that generates negative consequences to a project [5]. 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 work 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.

The complexity of both anti-patterns may worsen when usage dependencies go through an inheritance tree because of polymorphism. Next section 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 Figure 3. 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) has 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 become 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.
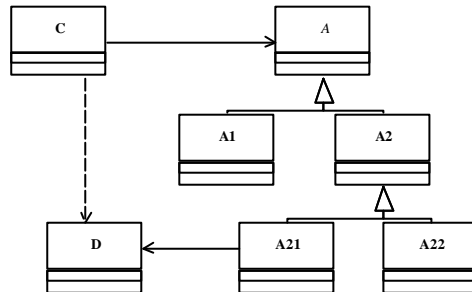

Figure 3 - Concurrent usage through an inheritance hierarchy

The testing model has thus to discriminate between up and down dependencies into an inheritance tree. Moreover, the testing model must not count sibling 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 one or more methods in 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. Figure 4 illustrates the two types of UML dependencies : an association between BOOKSTATE and BOOK classes and dependency BOOKEVENT and BOOK classes.*
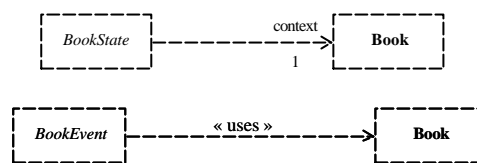

Figure 4 - association or dependency between classes

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 ith 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, Figure 5 illustrates two relationships between classes BOOKEVENT and BOOK. A «uses» dependency 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 Figure 6 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.
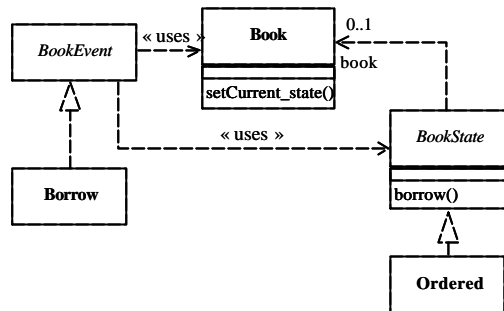
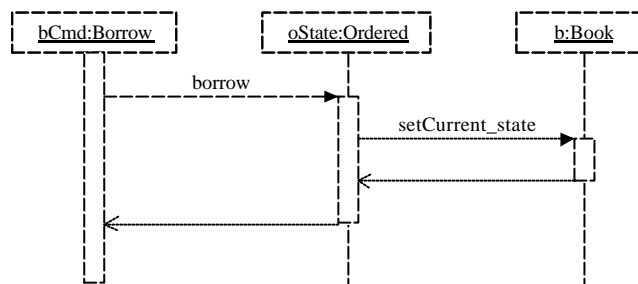**Figure 5 - Transitive relationship between BookEvent and Book through BookState**

Figure 6 - Sequence diagram illustrating a real indirect relationship between BookEvent and Book

Let us define now the notions of *class interaction* and *self-usage interaction*. These interactions are potential object interactions since they are detected from the class diagram which is only an abstract view of the software (and thus defines several possible implementations of object interactions). Indeed, the interactions detected at the design level can disappear or can be worsen when the design is refined into its implementation. 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 $A\ R_j\ B$, where R is a transitive relationship*
> *A self-usage interaction occurs around class A iff: $A\ R_i\ A$*

Figure 5 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 Figure 6 is associated to the class diagram of Figure 5, 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.*

From this test criterion definition, it appears that an estimation of the testing effort can be obtained from the class diagram, by detecting all class interactions and self-usages as well as their complexity. In [3], we have proposed a formal graph model that can be automatically derived from a UML class diagram and on which it is possible to detect all testability anti-patterns. We also proposed a complexity measurement of these anti-patterns based on the complexity of inheritance hierarchies involved in the anti-patterns.

Based on this graph model it is possible to identify the anti-patterns in the class diagram. It is then possible to focus on these particular parts to improve the design's testability. The second contribution of our work is to propose possible improvements on the design, to make more precise. 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.

Before introducing ways to improve the design's testability, next section illustrates the different class interactions and self-usages that can be identified on the class diagram of Figure 1. Then, a particular implementation (http://www.irisa.fr/triskell/results/Book/) is used to generate test cases that satisfy the proposed test criterion. This example also illustrates the difference between anti-patterns detected in the class diagram and the actual object interactions that have to be tested.

3.4. Example for test generation

This section introduces an example for test generation process using the adequacy criterion defined in previous section. The example is based on the class diagram of Figure 1. 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 Figure 2:

- 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/.

a) 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");
}
```

b)   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).
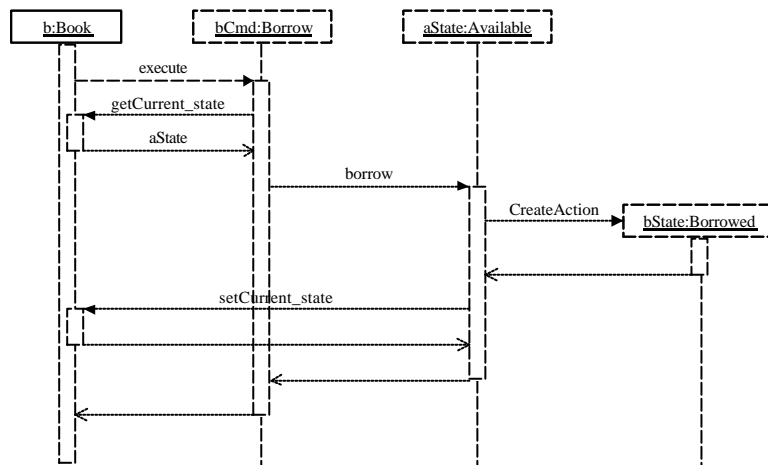
c)   CI interaction

Figure 7 - sequence diagram for test case n°3

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.changeState(new Borrowed());}
}
```

To summarize this third test objective, the following test case covers the interaction between BOOKEVENT and BOOK through two different paths and Figure 7 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 (Figure 1) according to the test criterion of previous section. This table 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. However, the complexity due to polymorphism is out of the scope of this paper.

table 1 - test report for the book manager sub-system

status	SU1	SU2 infeasible	CI
TC1	X		
TC2			X

4. Improving Design Testability

There are two possible ways to improve the design from a testing point of view. The first is to actually limit the number of anti-patterns either by reducing coupling between classes (limit relationships among classes) or by reducing the complexity due to polymorphism. The second way is to make the design more expressive to make the testability measurement on the class diagram more accurate and also constraint the implementation to make it more testable by design.

When it is possible, a way to improve testability and break inheritance complexity is to use of interfaces 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. 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 [6], 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 use of stereotypes modifies the identification of objects interactions w.r.t. the following properties.

> ***Assertion1 – 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).
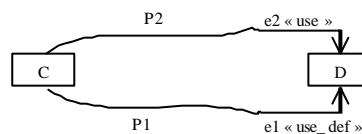

Figure 8 – a class interaction between C and D

Figure 8 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 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).

Since testing problems are too complex to be controlled on most overall designs, in previous work [7] we also studied particular generic microarchitectures widely used in the OO domain as basic refinement operators, the design patterns. These structures are interesting because they allow dividing the testability analysis of the design in small analyses of coherent sub-parts of the system's design. Another interesting point to notice is that patterns can be defined in terms of collaboration diagrams at the metamodel level of the UML: the elements for the patterns are defined in terms of roles as stated in [8]. This approach clarifies rigorously what a pattern is. Moreover, the testability stereotypes can be added on the metamodel specification of the design pattern. This insures that when the pattern is instantiated, the stereotypes are automatically added to the design.

5. Conclusion

An interesting technique to reduce the costs for testing is to take testability issues into account in design phases. This paper summarizes a work that proposes a testability analysis on OO designs, focusing in UML class diagrams as a reference model. We have identified testability anti-patterns as particular parts in the design on which a particular attention should be taken to limit testability problems in the implementation. The

testability is evaluated as the effort needed to generate test cases that satisfy a specific test adequacy criterion that consists at covering all interactions between objects. A second contribution of this work is to propose UML stereotypes to add information on relationships in the class diagram, that can help improve the testability of the design. These stereotypes also define expected properties on the implementation. Static checking can thus be used to check whether these constraints are satisfied in the implementation and thus limit the testing effort.

6. Bibliography

1. J.M. Voas and K. Miller, *Software Testability: The New Verification.* IEEE Software, 1995. **12**(3): p. 17 - 28.
2. R.V. Binder, *Design for Testability in Object-Oriented systems.* Communications of the ACM, 1994. **37**(9): p. 87 - 101.
3. B. Baudry, Y. Le Traon, and G. Sunyé. *Testability Analysis of UML Class Diagram.* In proceedings of *Metrics'02 (Software Metrics Symposium)*. Ottawa, Canada, June 2002. pp.54 - 63.
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Professional Computing. 1995. Addison-Wesley.
5. A. Correa, C.M.L. Werner, and 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.
6. S. Rapps and E.J. Weyuker, *Selecting Software Test Data Using Data Flow Information.* IEEE Transactions on Software Enginnering, 1985. **11**(4): p. 367 - 375.
7. B. Baudry, Y. Le Traon, G. Sunyé, and J.-M. Jézéquel. *Towards a Safe Use of Design Patterns for OO Software Testability*. In proceedings of *ISSRE'01 (Int. Symposium on Software Reliability Engineering)*. Hong-Kong, China, November 2001. pp.324 - 329.
8. G. Sunyé, A. Le Guennec, and J.-M. Jézéquel. *Design Pattern Application in UML.* In proceedings of *ECOOP'00*, June 2000. pp.44 - 62.