Measuring and Improving Design Patterns Testability

Benoit Baudry[*], Yves Le Traon[*], Gerson Sunyé[**] and Jean-Marc Jézéquel[*]
[*] IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France
{Benoit.Baudry, Yves.Le_Traon, Jean-Marc.Jezequel} @irisa.fr
[**] IRIN - Faculté des Sciences et Techniques de Nantes, 44322 Nantes cedex 03, France
sunye@irin.univ-nantes.fr

Abstract

This paper addresses not only the question of testability measurement of OO designs but also focuses on its practicability. While detecting testability weaknesses (called testability anti-patterns) of an OO 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, this paper investigates solutions integrated to the 00 process. We focus on the design patterns as coherent subsets in the architecture, and we explain how their use can provide a way for limiting the severity of testability weaknesses, and of confining their effects to the classes involved in the pattern. Indeed, design patterns appear both as a usual refinement instrument, and a cause of complex interactions into a class diagram – and more specifically of testability anti-patterns. To reach our objective of integrating the testability improvement to the design process, we propose first a testability grid to make the relation between each pattern and the severity of the testability anti-patterns, and we present our solution, based on a definition of patterns at metalevel, to automate the instantiation of patterns constrained by testability criteria.

1. Introduction

Testability is a quality factor that is useless if it is not available early in the life-cycle. 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. A measurement of the design's testability is essential, but must be coupled to practical guidelines for helping the designer improve the design's testability.

The contribution of this paper concerns both a given metric and the practical way to apply it in the usual OO design process. While in [3], we presented a testability measurement and the graph model to capture it from a UML class diagram, we had not studied yet the practical applicability of such a measurement on final OO systems. The factor we measure corresponds to undesirable configurations in the class diagram we call *testability anti-patterns*. When using our measurement on final designs, we remark it is very hard for a designer, who is not necessarily a specialist of testability measurement, to take the right decisions for improving this design, due to its complexity. We are convinced that this problem is met by most of the "users" of OO measurements, and this paper investigates solutions integrated to the OO process. Ideally, a designer should not have to deal both with functional objectives and with testability, or other non-functional properties. So, the global motivation that is at the core of this paper is to show how to integrate testability improvements into the usual design process.

Until recently, the final OO architecture often appeared as a complex set of interacting classes with no logical subsets emerging from the global design. However, thanks to the UML standard, systematic methodologies [4, 5], now offer a decomposition approach for the architecture or guidelines to deal with evolution. These methodologies help design object-oriented software as a sequence of refinements, from initial analysis to the implementation. Specifically, design patterns [6] may serve as a basis for such a refinement. Starting from an analysis class diagram, design patterns help the designer in reusing design solutions to solve problems in a particular context, and thus transform the diagram into a more implementation-aware one. Design patterns then correspond to subsets in the class diagram, and can be considered as intermediate structures between the overall architecture and the single class. This system decomposition provides an interesting solution, at a local level, for problems that are too complex at the global level.

Design patterns appear both as a usual refinement instrument, and a cause of complex interactions into a class diagram – and more specifically of testability anti-patterns. To reach our objective of integrating the testability improvement to the design process, we consider that each refinement of a class diagram (due to the application of one or several design patterns) must lead to another testable class diagram or to a decision of introducing a testability weakness.

This work has been partially supported by the CAFÉ European project. Eureka Σ! 2023 Programme, ITEA ip 004

The initial answer to this objective consists in a catalogue, a testability grid, that establishes the relationship between a chosen design pattern, the parameters that impact the testability and the corresponding value of testability. This grid offers a guideline for a design decision, but it doesn't help for improving the testability. We thus explain a "by hand" solution, that consists in studying each design pattern and add some information to clarify the functional usage of some class diagram dependencies. Most of the testability issues disappear with a clever use of three specific stereotypes we introduce, as in correspondence with the dataflow model for procedural program [7]: «create», «use_consult» and «use_def». Using these stereotypes guarantees that the implementation will not introduce actual objects interactions and side-effects. A static verification can be performed on the code, to check that stereotypes are well implemented.

However, this initial answer is incomplete without a way of automating the improvements. Indeed, the ideal way for a feasible design for testability approach is to automate the insertion of the right stereotypes when applying a design pattern. The difficulty is to describe design patterns in a generic way, so that they can be applied automatically on any valid UML model. The application of a design pattern is a transformation on a UML model, that can be defined in the UML metamodel (as [8] introduced this approach). The application of a design pattern is an instance of its specific metamodel and is thus constrained by the features defined at this level. One of the other advantages of a metalevel definition is that the solution is compatible with any CASE tool based on the UML metamodel. In this paper, we illustrate this application of design patterns constrained for testability.

The paper is organized as follows. Section 2 opens with a summary of the testability measurement presented in [3], and the stereotypes we propose to improve the testability of a class diagram. Section 3 introduces design patterns, and pattern-based software design, and section 4 illustrates the testability analysis on two examples, then summarizes testability issues for the application of design patterns. In section 5, we study abstract representations for design patterns (within the UML metamodel), to automatically apply design patterns in a testable way.

2. Testability model and Test Criterion

Testability is a quality factor for programs, or program architectures that relates to the easiness for testing a piece of software[2, 9, 10]. Both the designer and the project manager can use it for different purposes. It offers a tool for the software designer who wishes to identify hard-to-test systems while still at the design stage. The project manager can use this measure to find a trade-off in terms of cost between solutions based on different designs and different testing methods. The testability measurement is influenced by three parameters: controllability, observability [9] and the global test cost. This work focuses on the latter to estimate the testability of UML class diagrams.

> **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 finding out the right test data and the difficulty of deciding on the validity of the run results.*

In [3], we proposed a test criterion for an object-oriented system, based on particular configurations among classes. The estimation of the effort needed to verify this criterion is an estimation of the global test effort. This section recalls the criterion as well as the configurations it aims at covering. Two configurations have been identified and they are called *testability anti-patterns*, as they describe patterns that should be avoided for a testable design.

2.1. Testability Anti-patterns and test criterion

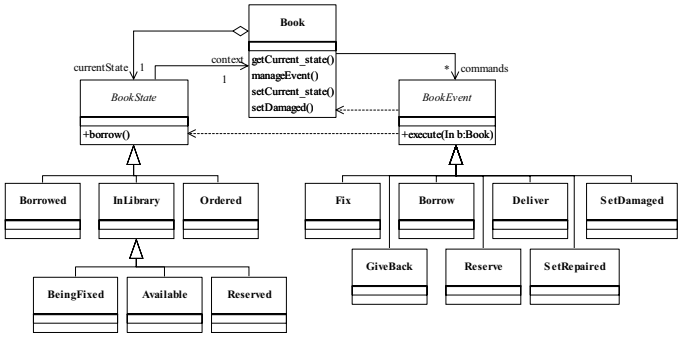

Figure 1 - book management system

The test criterion we propose aims at covering particular configurations that can appear on a UML class diagram. These configurations are illustrated based on the class diagram of Figure 1 representing a book management system. This architecture is a typical object-oriented design, as 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 basis for testing 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.

Two weaknesses for testability appear on this class diagram: 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 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 [11]. 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 represents a bad solution to problems that arise when software is being developed in a particular context. It corresponds to a configuration in the class diagram, which increases the testing effort needed to satisfy the test criterion.*

Class interaction. *A class interaction occurs, in a class diagram, from a class A to a class B if there are two or more paths going from A to B.*

Self usage interaction. *A self usage interaction occurs, in a class diagram, around a class A if there is one or more cycle(s) from A to A.*

It has to be noted that the "severity" of an anti-pattern increases with the number of paths (or cycles) involved. In the testability grid (Table 1 in section 4.3), we present the number of paths predicted for the anti-patterns detected at the application of each main design pattern.

MAIN CAUSES

The anti-patterns may occur:

- when concepts – represented by classes – are naturally interconnected (Class interaction) or reflexive (Self-Usage).
- when local refinements introduce specific dependencies that create global cycles in the overall design.
- when the coupled parts of the design include inheritance dependencies.

SYMPTOMS

- on the design (from the most general to the most specific) : high coupling, cycles of dependencies (Self-Usage), large number of possible interacting objects (Class interaction).
- at the code level : prohibitive number of interleaved execution paths involving the same objects (Class interaction), inconsistent object states, side-effects (Class interaction and Self-Usage).

CONSEQUENCES

- on the design : Difficulty to predict precisely the objects responsibilities for a given execution. Hard to specify what the test cases should cover from the design. Large number of test purposes.
- on the code: prohibitive testing effort. Large number of infeasible test purposes, large number of test cases and oracles.

SOLUTIONS (to be detailed in the following)

Dealing with a whole final design is in general impossible: control the testability evolution during refinement steps. This implies the following sub-solutions:

- measurement of the severity of anti-patterns,
- identifying the ambiguous links in the architecture and add constraints on the specification to reduce the possible spectrum of incorrect implementations.
- automate the insertion of constraints when using classical design solutions (design patterns).

The exact number testability anti-patterns is difficult to determine with a simple observation of the design. Moreover, the complexity for testing increases when inheritance hierarchies are involved in such anti-patterns. In [3], we propose a graph model, which can be derived form a UML class diagram, and from which it is possible to compute all anti-patterns that affect testability, as well as the complexity due to inheritance.

Both class and self usage interactions are potential interactions since they are detected from the class diagram which is only an abstract model for the system. Thus, we also define the notion of object interaction, which is an interaction between objects of the running system.

Object interaction. *An object interaction occurs in a system if an object o_1 uses an object o_2 through two or more paths, or if o_1 uses itself.*

The test criterion can be defined, based on the notions of class and object interactions.

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

The task of producing test cases/reports is impossible if the number of class interactions is high. We study how to the limit these interactions by improving the design. Indeed, the design must be as close as possible to the code. Since the number of anti-patterns is an upper bound of the number of object interactions, we recommend putting additional information on the design that would reduce the number of class interactions. These additional pieces of information are design constraints for the programmer (e.g. expressed using UML stereotypes, or the OCL): 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.

2.2. Improving a class diagram for testability

Improving the 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 earlier, a solution may consist in clarifying the design, so that the code can be as close as possible to what the designer wants. The UML offers two ways to add information on a class diagram: the Object Constraint Language (OCL), or the definition of *stereotypes*. The OCL [12] is a formal language to express constraints on UML diagrams and can be used to describe invariant on classes, and pre and post conditions on methods. We do not present this language in this paper.

We define several stereotypes that specify the semantic of links involved in class interactions (association, dependency, aggregation, composition). Thanks to these additional specifications, the programmer should avoid implementing an object interaction. The stereotypes introduced here are analogous in some way to data flow testing criteria for classical software [7] that identify "definition" and "use" of variables in a program. This classical testing model aims at determining the data flow, the "life line" of variables at unit level.

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.

Objects interaction: *Let P1 and P2 be two paths from class C to class D, defining a class interaction between C and D. An objects interaction exists* iff

- *the last edge of path P1 and the last edge of path P2 have associated stereotypes «use» or «use_def» (Figure 2).*

Let P be a path from class C to itself, defining a self-usage interaction for C. An object interaction exists iff *:*

- *the last edge in the cycle has either «use» or «use_def» stereotype.*

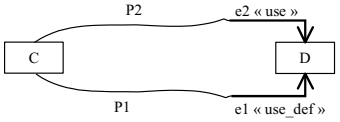

Figure 2 – a class interaction between C and D

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).

Solving testability problems on a whole system design can be very difficult because of the numerous classes that can be involved in the anti-pattern. In the following, we focus on design patterns as consistent and well-defined subsystems, and study these micro-architectures to solve testability issues at local levels. Section 3 gives an example for pattern-based OO design and sections 0 and 5 concentrate on solving testability problems for design pattern applications.

3. Designing by pattern crystallization

This sections starts with a definition of a design pattern, and illustrates how the application of patterns corresponds to a crystallization phenomenon. Thanks to this technique, it is possible to distinguish coherent subsets in the class diagram. These subsets correspond to design patterns applications, and allow the designer to evaluate the testability of a design at a local level. This decreases the complexity of the analysis, and of the improvements that may be necessary.

Design patterns. *Design patterns represent solutions to problems that arise when software is being developed in a particular context. Design patterns can be considered as reusable microarchitectures that contribute to an overall system architecture; they capture the static and dynamic structures and collaborations among key participants in software designs.*

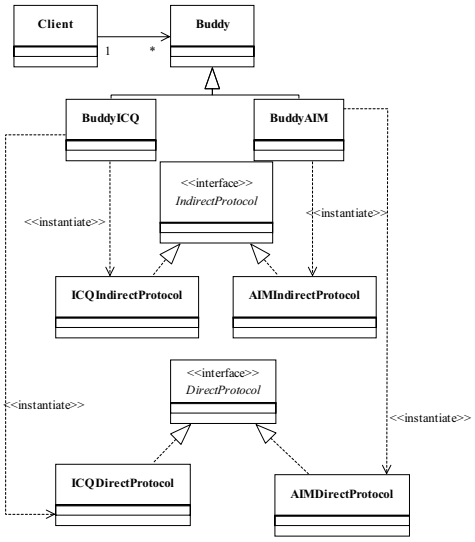

Figure 3 - An Instant-Messaging client

Figure 3 shows an early object-oriented design (analysis stage) for an instant messaging client. It allows several persons to chat, and several protocols may be used (ICQ, AIM). There are two central classes in this architecture, Client and Buddy. Both classes can be either in a connected or a non-connected state. Depending on the state of a buddy instance of Buddy, an instance of the Client is connected to buddy via a direct or indirect protocol. Figure 4 illustrates a possible final detailed design after several refinement steps, showing design patterns instantiations in ellipses as per the UML standard.

The architecture of Figure 4 is a typical object-oriented design obtained after crystallization stages. A crystallization stage involves adaptation of a design pattern [6] to a class diagram. This approach is a widely used methodology for steering of an initial analysis diagram into a more implementation-aware level. After the application of a design pattern, main analysis classes remain on the diagram, but also new classes and links appear and some association relationships are deleted. In this example, the analysis class diagram was modified through three independent refinement steps corresponding to the adaptation/combination of one Abstract Factory and 2 State design patterns.

The new links (inheritance, associations, etc.) and classes, introduced when refining the design using design patterns, seem to make the design very hard to test. Each class may potentially interact with any other. However, the development methodology (crystallizing the design patterns from an initial analysis) actually allows us to consider the whole design as a composition of microarchitectures, instead of as a monolithic set of interconnected classes. As a result, the overall complexity may be decomposed into a combination of the microarchitecture complexities, and the testing task may be simplified. Indeed, once subsets are identified in the design, several testing problems can be solved more easily at their local (microarchitecture) level, than at the system level.

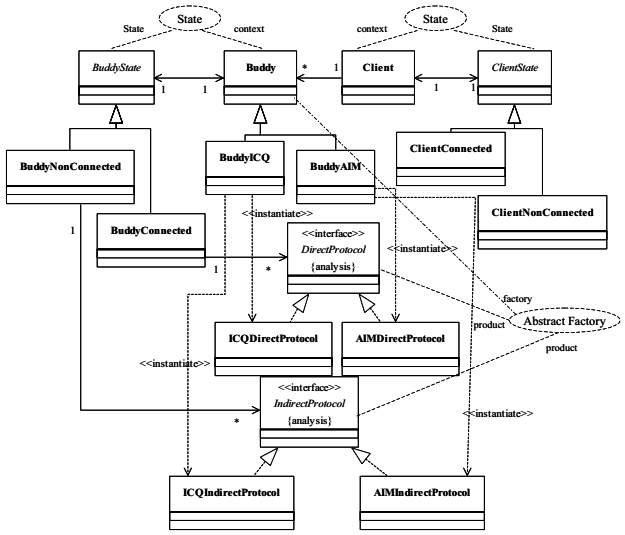

Figure 4 - An Instant-Messaging client (a possible "final" design)

4. Testability of Design Patterns

We detail the testability analysis for two classical design patterns, namely the Builder and the Composite ones. It shows how the testability grid is obtained and underlines the possible solutions for suppressing the detected anti-patterns.

4.1. Builder

Figure 5 displays an application of the Builder Design Pattern taken from [13]. The intent of this pattern is to "separate the construction of a complex object from its representation so that the same construction process can create different representations". It is very similar to the Factory pattern. In our case, we consider a maze game, and the MAZEGAME class delegates the creation of a particular maze (MAZE class) to a Builder and then manipulates it. A class interaction appears from MAZEGAME to MAZE: there is a direct path form MAZEGAME to MAZE, and there is path going through the DIRECTOR, MAZEBUILDER and STANDARDMAZEBUILDER classes. Looking at Table 1, in this particular case, there is only one concrete product and

two concrete builders, which lead to one anti-pattern, between MAZEGAME and MAZE. Due to the inheritance hierarchy, the number of possible paths in this anti-pattern is 4 (it would have been 8 if the second Builder COUNTINGMAZEBUILDER was connected to the products).

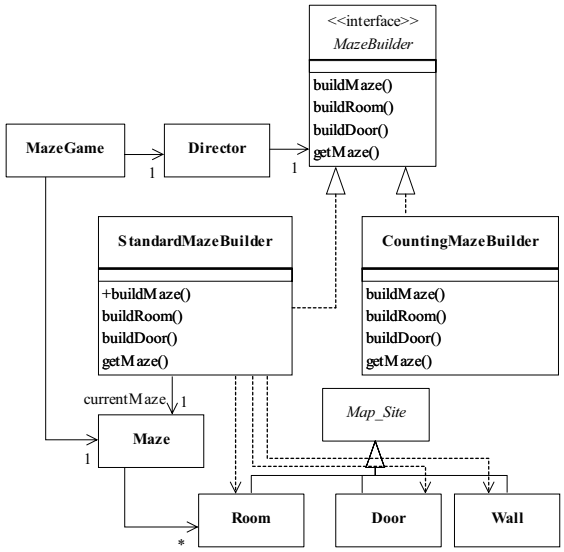

Figure 5 - An application of the Builder design pattern

This complexity of an anti-pattern, as a number of possible paths involved in it, can be generalized and predicted for each design pattern. Indeed, for one client class (here the MAZEGAME), one abstract builder class (MAZEBUILDER), let n be the number of concrete builders (here STANDARDMAZEBUILDER and COUNTINGMAZEBUILDER) and p the number of products (ROOM, DOOR, WALL, MAZE), we have the following relation between the number X of paths in the anti-pattern (a class interaction) and the parameters : $p \leq X \leq n*p$.

Indeed, in the worst case, each concrete builder creates each type of concrete product (n*p paths), while there is always a direct dependency linking the client to each product (1 path). In the best case, each concrete product is created by only one builder. By using stereotypes, these anti-patterns can be avoided.

To test the application of the Builder pattern, we must check that the delegation from the MAZEGAME to the builder creates all objects and *does not do anything else*. If the design pattern application is well implemented, the MAZEGAME class uses creation methods of concrete products through the MAZEBUILDER inheritance hierarchy and uses other methods directly calling the concrete MAZE objects. In that case, there is no interaction at all since the concurrent paths between the MAZEGAME and the MAZE are used for different purposes.

A solution to delete this class interaction would be to express implementation constraints on the design and to specify clearly the delegation. For example, the association from MAZE to DIRECTOR and MAZEBUILDER could be labeled with a «use_create» UML stereotype. This informs the developer that MAZEBUILDER uses only creation methods of class MAZEBUILDER.

4.2. Composite

The Composite aims at letting clients treating individual objects and compositions objects uniformly: in this case the FILEMANAGER class will be able to apply the same treatments to FILE objects and DIRECTORY ones, the last being a composition of other DIRECTORY and FILE objects.

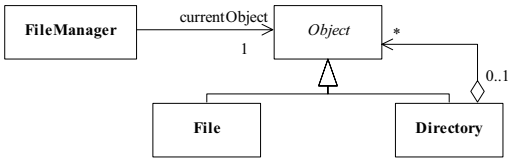

Figure 6 - An application of the Composite design pattern

Figure 6 displays an application of the composite design pattern, to describe the structure of a simple file manager. This manager deals with object that can be either files or directories, and a directory is composed of multiple objects. There is one self-usage interaction around the FILE class. The number of paths involved in the detected anti-pattern is 2, corresponding to the 2 concrete classes that are involved in the cycle.

To generalize this counting, we consider a typical application of this pattern with one client (here FILEMANAGER), one abstract component class (OBJECT) and n concrete component classes (FILE and DIRECTORY). In this case the predicted number of paths involved in the anti-pattern is equal to n. By the same way, the complexity of anti-patterns can be predicted for each design pattern: the results are summarized in the testability grid presented in the following section.

To improve the testability of the class diagram shown Figure 6, a first solution is to refactor the OBJECT class as an interface class: this avoids links from children classes to OBJECT. If only leaf classes in the inheritance hierarchy (classes that have no descendants) are concrete classes, the complexity of the interaction going through this hierarchy is reduced since there are no interactions between classes in the hierarchy.

Another solution is to express a constraint on the model that corresponds to the semantics of the file manager, and more generally to the application of the composite design pattern: a directory can contain several other objects, but it does not contain itself. So if this constraint is well

implemented, the class interaction is still present on the class diagram and in the derived graph, but it will never become an object interaction. This constraint can be expressed with the OCL (Object Constraint Language) on the UML class diagram in the following way:

context Directory **inv**:
not (object->includes(self))

4.3. The testability grid for design patterns

To guide designers, we propose the grid of Table 1 to estimate how risky a 'naïve' use of a design patterns may be (i.e. without specifying precisely the kind of associations between classes into use-def/use-consult/create). We consider the pattern instantiations proposed by Gamma [6] since we are convinced most designers refer to them, even if a pattern is first a generic concept. It allows us to parameterize each pattern as a function of the main elements it involves. For sake of conciseness, we summarize behaviors for fundamental design patterns [14] that are: Abstract Factory, Bridge, Builder, Composite, Decorator, Flyweight, Proxy, Iterator, Mediator, Memento and State. We remark that there are two categories of design-patterns, those that imply only Class-interactions anti-patterns, and the others only Self-usage ones: a transversal classification of the patterns appears, that is independent from their functional objectives. In terms of testability, the grid shows that the less testable patterns are Mediator and Visitor (a quadratic increasing of the complexity in function of the parameters). Mediator is very similar to the Observer pattern (so Observer is not detailed), which testing has been exhaustively studied by Mc Gregor in [15]. The Visitor pattern is especially known to be difficult to test because of an extensive use of polymorphism and an implementation based on double dispatch.

Table 1 – Testability issues when applying design patterns

Design Pattern	Number of participants	# paths in a class interaction	# cycles in a self usage interaction
Abstract Factory	1 client 1 abstract factory class n concrete factory m abstract products p concrete products	$p \leq X \leq n*p$	no
Bridge	parameters do not impact on testability	no	no
Builder	1 client 1 abstract builder class n concrete builders p products	$p \leq X \leq n*p$ (same type of interaction as Abstract Factory)	no
Composite	1 client 1 abstract component class n concrete component classes	no	n
Decorator	1 interface 1 component class 1 abstract decorator class n concrete decorator classes	no	$1 \leq X \leq n$
Flyweight	Like abs. fact.	Like abs. fact.	Like abs. fact.
Iterator	1 client n ConcreteAggregate n ConcreteIterator	no	2*n (n from client to ConcreteIterator and n from client to ConcreteAggregate)
Proxy	1 subject abstract class 1 proxy 1 real subject	no	no
Chain of responsibility	1 client 1 handler abstract class n abstract handler classes	no	n

Mediator	1 mediator abstract class 1 colleague abstract class n colleague concrete classes m mediator concrete classes	no	m*n
Memento	1 memento 1 originator 1 caretaker	no	no
State	1 context class 1 root abstract state class n concrete state classes (Implicit delegation link from concrete state classes to context class)	no	n
Visitor	n visited elements (ConcreteElement) p visitors (ConcreteVisitor)	no	n*p

Not all the patterns can be deleted using the stereotypes "create" and "use" already presented. For example, the Composite design pattern self-usage anti-pattern is only deleted if we can specify the constraint expressed in section 4.2. It would be a very tedious task for a designer to do that every time he uses a Composite pattern. Next section illustrate how expressing this constraint on the metamodel definition of the pattern allows to automate the insertion of this constraint. Moreover, in the case of the State design pattern, we need to express a much more complex constraint to specify the precise roles of the methods (separating the one that manages the current state from the others that must not modify the current state): this cannot be done directly in the model with the OCL language. However, this constraint can be defined in the metamodel, since the notion of method is explicit at this level.

5. Defining Testability Constraints for Design Patterns at Meta-Level

We present now how testability constraints can be automatically inserted when a design pattern is instantiated. We use the notion of parameterized collaboration and illustrate its application on the Factory design pattern, that is very close to the Builder pattern.

Object Constraint Language is the constraint language developed by the OMG (Object Management Group) for the UML. At a first sight, it seems the natural way of adding testability constraints to a UML design. Unfortunately, we found that such rules are very hard to write in OCL and may lead to an unrealistic solution. For example, a one page long OCL expression is needed to tell that a delegation should only implement creation links and nothing else. Another option is to define the pattern applications 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 application is, and embeds the expected testability properties at a generic level. Automatic verification tools can be produced then to check whether a pattern is safely implemented at code-level.

5.1. Collaboration Diagrams for Design Patterns Representation

In [6], Gamma et al have used the OMT graphical notation (class and dynamic diagrams) to represent the structure of a design pattern. While these diagrams can accurately represent an occurrence (or instance) of a pattern, they are not able to represent the pattern structure itself, which is expressed in terms of "roles", played by model elements (e.g. classes, attributes, associations, methods).

The use of parameterized collaborations in UML, which are similar to "frameworks" in Catalysis, is a promising approach to describe the structure of a pattern. In this approach, one can represent the role that should be played by a pattern participant, instead of the participant itself. In Figure 7, we use a parameterized collaboration to represent the Abstract Factory design pattern, where two roles can be identified (Factory and Product). When this collaboration is used, i.e. each role is linked to an effective class, a dependency relation is created between these classes.

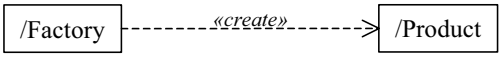

Figure 7 - Parameterized Collaboration

However, as stated in [8], collaborations still present several lacks and can not precisely represent design patterns. In this particular case, for instance, designers are not able to specify if these are the roles of individual classes or roles of class hierarchies. The designer cannot either specify that the Factory should own a "creator" method. Moreover, roles are limited to classes and

associations, whereas patterns are also composed by attributes and methods.

A possible workaround for these lacks is to use collaborations to extend the UML meta-model. The idea is that pattern constraints may be attached at this meta-level so that the right stereotypes will be automatically generated each time a designer instantiates a pattern. We illustrate this approach on the Factory design pattern.

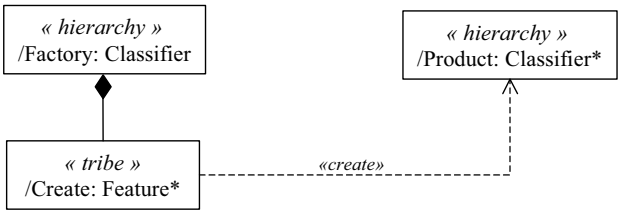

Figure 8 - The Abstract Factory Design Pattern

Figure 8 presents the same pattern, using the notation, proposed in [16], where patterns are described as meta-model level collaborations, completed by constraints. In this representation, the pattern is composed of three roles: Factory, Product and Creator. The first role is a «hierarchy», i.e. a set of classes linked by a generalization relationship. The second one is a set of hierarchies.

Finally, the third role describes a set of «tribes». A «tribe» [17] is a set of features sharing a common signature. Each feature is owned by an element of the Factory hierarchy. A constraint is attached to the Creator role: it must perform a single action, the instantiation of a Product (this constraint is reified by the «create» link). This implies that the multiplicity of Creator tribes is equivalent to the multiplicity of Product hierarchies.

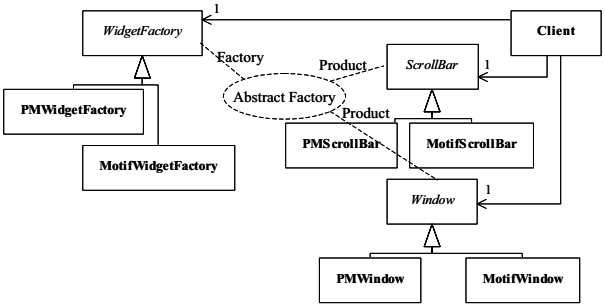

Figure 9 - An instance of the Abstract Factory design pattern

An instance of this pattern is presented in Figure 9. The Window and ScrollBar class hierarchies play the role of Products, and the WidgetFactory class hierarchy plays the role of Factory. There are two clans of creation methods in the WidgetFactory hierarchy (CreateScrollBar() and CreateWindow()). This Factory instance in combination with one of the meta-level representations of Figure 7. or Figure 8 implies that any dependency between a WidgetFactory (that instantiates the Factory role) and a Window or Scrollbar (that instantiates the Product role) is stereotyped «create». Thus, the pattern instance specification is automatically completed in terms of testability to avoid the anti-pattern from Client to Product. The following task is a verification one (that can be statically performed by a CASE tool) and consists in checking whether the «create» constraint is well implemented at code level.

6. Related work

A large number of measures have been proposed to evaluate the quality of object-oriented designs [18], one of them is coupling. The coupling measures the strength of the relationship between two 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 [19]. As stated in [3], this paper proposed first a mapping of a coupling measurement to precise modeling elements of the UML.

Concerning the relationship between coupling and the testing effort, studies (in terms of testability in [1] and test criteria in [20]) can be found for the coupling between object (CBO) measure. The difference with our work is that paths are considered independently, and all of them are measured and have to be covered. Here, we concentrate on particular paths that contribute to interactions in the overall system. Concerning design patterns, very few work ([21, 22]) have studied the use of design patterns and its implication on software quality factors (maintenance, reliability). However, we consider this research direction as one of the most promising way for dealing with refinement and evolution of an OO software.

7. Conclusion

In the case of OO designs, control flows are generally not hierarchical, but are diffuse and distributed over the whole architecture. From a testing point of view, such control flows may be hard to test, especially when dynamic binding and polymorphism are involved. We introduce the concept of a "testability anti-patterns," when potentially concurrent client/supplier relationships between the same classes along different paths exist in a system. The notion is topological and correspond to a detectable configuration in the class diagram. This paper discussed two configurations of an OO design, called anti-patterns, that can weaken its testability. Since testing problems are usually too complex to be fully controlled at the global level, we discussed particular design patterns microarchitectures, widely used in the OO domain, as possible basic refinement operators. We illustrated how

testing risks might be avoided. The two risk mitigation techniques we used are :

- a guideline of the risk for applying a pattern called the testability grid,
- and design refinement constraining.

For the second, we studied how the application of a pattern can be automatically constrained by its representation at meta-level.

Bibliography

[1] R.V. Binder, "Design for testability in object-oriented systems". Communications of the ACM. vol. 37(9): p. 87-101, September 1994.

[2] J.M. Voas and K. Miller, "Software Testability: The New Verification". IEEE Software. vol. 12(3): p. 17-28, May 1995.

[3] B. Baudry, Y. Le Traon, and G. Sunyé, "Testability Analysis of UML Class Diagram". In proceedings of *Software Metrics Symposium*, Ottawa, Canada, pp. 54-63, June 2002.

[4] D.F. D'Souza and A.C. Wills, "Object, Components and Frameworks with UML, The Catalysis Approach". Object Technology, Addison-Wesley, 1998.

[5] R. France and J.M. Bieman, "Multi-view software evolution: a UML-based framework for evolving object-oriented software". In proceedings of *ICSM'01*, Florence, Italy, pp. 386-95, November 2001.

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

[7] S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information". IEEE Transactions on Software Enginnering. vol. 11(4): p. 367-375, April 1985.

[8] G. Sunyé, A. Le Guennec, and J.-M. Jézéquel, "Design Pattern Application in UML". In proceedings of *ECOOP'00*, pp. 44-62, June 2000.

[9] R.S. Freedman, "Testability of Software Components". IEEE Transactions on Software Enginnering. vol. 17(6): p. 553-564, June 1991.

[10] Y. Le Traon, F. Ouabdessalam, and C. Robach, "Analyzing testability on data flow designs". In proceedings of *ISSRE'00*, San Jose, CA, USA, pp. 162-73, October 2000.

[11] 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*, pp. 336-352, June 2000.

[12] OMG, "Object Constraint Language Specification" http://www.omg.org/docs/ad/97-08-08.pdf

[13] V. Sawant, "Design Patterns using Java" http://www.cs.unc.edu/~vivek/home/academic.html

[14] E. Agerbo and A. Cornils, "How to preserve the benefits of design patterns". In proceedings of *OOPSLA'98*, Vancouver, BC, Canada, pp. 134-43, October 1998.

[15] J.D. McGregor, "Test Patterns: Please Stand By". Journal of Object Oriented Programming. vol. 12(3): p. 14-19, June 1999.

[16] A. Le Guennec, G. Sunyé, and J.-M. Jézéquel, "Precise Modeling of Design Patterns". In proceedings of *UML'00*, pp. 482-496, October 2000.

[17] A.H. Eden, "Precise Specification of Design Patterns and Tool Support in their Application". University of Tel Aviv. 1999.

[18] M. Shepperd, "Object-Oriented Metrics: an Annotated Bibliography" http://dec.bournemouth.ac.uk/ESERG/bibliography.html

[19] L. Briand, S. Morasca, and V.S. Basili, "Property-based Software Engineering Measurement". IEEE Transaction on Software Engineering. vol. 22(1): p. 68-86, January 1996.

[20] R.T. Alexander and J. Offutt, "Criteria for Testing Polymorphic Relationships". In proceedings of *ISSRE'00 (Int. Symposium on Software Reliability Engineering)*, San Jose, US, pp. 15-23, October 2000.

[21] J.M. Bieman, D. Jain, and H. Yang, "OO design patterns, design structure, and program changes: an industrial case study". In proceedings of *ICSM*, Florence, Italy, pp. 580-589, November 2001.

[22] L. Prechelt, B. Unger, W.F. Tichy, P. Brössler, and L.G. Votta, "A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions". IEEE Transactions on Software Enginnering. vol. 27(12): p. 1134-1144, December 2001.