Towards a 'safe' use of design patterns to improve oo software testability

Benoit Baudry, Yves Le Traon, Gerson Sunyé, Jean-Marc Jézéquel

Towards a 'Safe' Use of Design Patterns to Improve OO Software 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, Gerson.Sunye, Jean-Marc.Jezequel} @irisa.fr

Abstract

Design-for-testability is a very important issue in software engineering. It becomes crucial in the case of OO designs where control flows are generally not hierarchical, but are diffuse and distributed over the whole architecture. We introduce the concept of a "testing conflict" when potentially concurrent client/supplier relationships between the same classes along different paths exist in a system. Such conflicts may be hard to test, especially when dynamic binding and polymorphism are involved. We describe the conflicts using topological class configuration diagrams. An overall architecture is represented as a combination of the initial design and several patterns. 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 complexity of testing for conflicts, and of confining their effects to the classes involved in the pattern.

1. Introduction

Testing is often a very costly part of a software life cycle. Any technique that improves a software design at an early stage, can have highly beneficial impact on the final testing cost and its 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. The general goals are: 1) identification of those parts of an OO software architecture which may need more testing due to potential problems arising from undesired object and parameter interactions, and 2) presentation of a way of decomposing/mastering the testability weaknesses using design patterns as elements in OO design refinements.

This question of testability [Voas91] has been revived with the object-orientation ([Binder94, Voas96]). Object-orientation is now in widespread use by the software industry, despite the fact that the technology is not mature enough from the testing point of view. While in an OO context, new testing challenges arise due to complex interactions and couplings among classes, some classical testing problems, such as covering control flow graphs, may disappear.

Until recently, the final OO architecture often appeared as complex set of interacting classes with no logical subsets emerging from the global design. However, thanks to the UML standard, systematic methodologies, such as Catalysis^{SM} [DSouza98], now offer a decomposition approach for the architecture. These methodologies help design object-oriented software as a succession of refinements, from an initial analysis to the implementation. Specifically, design patterns [Gamma95] 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.

At the system level, testing is usually of the “black-box” variety, and is often not formalized. When it is formalized, it may require strong (and possibly unrealistic) assumptions concerning the completeness of the underlying behavioral and dynamic models [Binder99, Briand01]. An effective OO testing strategy should be able to identify the trouble spots, and then resolve the problem by taking into account the complexity of the control flows over the whole architecture.

In this paper, we focus on testing problems that appear at system level due to interactions among classes and due to polymorphism. These problems are very difficult to solve globally because of the great number of classes involved, and because of the coupling among these classes. But, the same problem may be much easier to tackle at a sub-system level. We focus on two specific testing difficulties. Those arising from:

- static design ambiguities, and
- non-statically decidable concurrent use of the same objects by a common client.

Both issues are mixed in practice, and both lead to the same kind of testing problems. We call them “testing conflicts” or contradictions. These conflicts correspond to specific topological configurations that can be detected from the class diagram. We discuss how these configurations can be made less complex, or even completely avoided at a sub-system level. This includes using informal analyses and advice on how to improve

class diagrams, and inclusion of explicit constraints for the implementation.

Section 2 offers a brief introduction to pattern-based object-oriented design, and its relationship to testing of object-oriented software. Section 3 discusses testing problems that can arise from poor design refinement and implementation of an object-oriented architecture. Section 4 illustrates the concept of testing conflicts using Design Patterns and proposes a solution that avoids testing conflicts.

2. Pattern-based construction

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.*

2.1. Designing by pattern crystallization

Figure1 shows an early object-oriented design (analysis stage) for an instant messaging client. 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 2 illustrates a possible final detailed design after several refinement steps, showing design patterns instantiations in ellipses as per the UML standard.

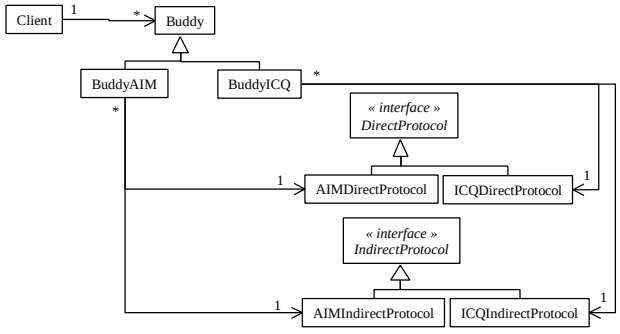

Figure 1 - An Instant-Messaging client (analysis diagram)

The architecture of figure 2 is a typical object-oriented design obtained after crystallization stages. A crystallization stage involves adaptation of a design pattern [Gamma95] 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. The questions we will try to answer using case-based illustrations are:

- What are the testability improvements when using design patterns?
- Can the testability problems be localized (confined) to the diagram subset that corresponds to the application of a design pattern?

We discuss the answer to the first question in the context of a testability comparison between isolated use of the design pattern "State" and the classical/"functional" implementation of a state machine using a single class. We discuss the second question - definition of rules/constraints on the design which avoid testing conflicts - through a solution that attaches constraints to a design such that the implementation is testable.

2.2. Testability of the State Design Pattern

We distinguish between object oriented and "functional" programming through the effort dedicated to the design. What we call object-oriented software, is software for which most effort is put on designing the architecture, the modular (class) decomposition, the coupling between classes (using as much as possible particular object-oriented constructs, such as inheritance and polymorphism). We illustrate the differences via a "functional" and an "OO" implementation of a finite state machine.

The example is taken from [Jézéquel99]. It implements the state machine associated with the `Mailer` class (figure 3). This class defines a public interface for sending and receiving messages, regardless of the connection status with the network. When the network connection is available, the messages are simply forwarded to it. If it is disconnected, outgoing messages are buffered in the mailer for subsequent transmission when the network connection is restored.

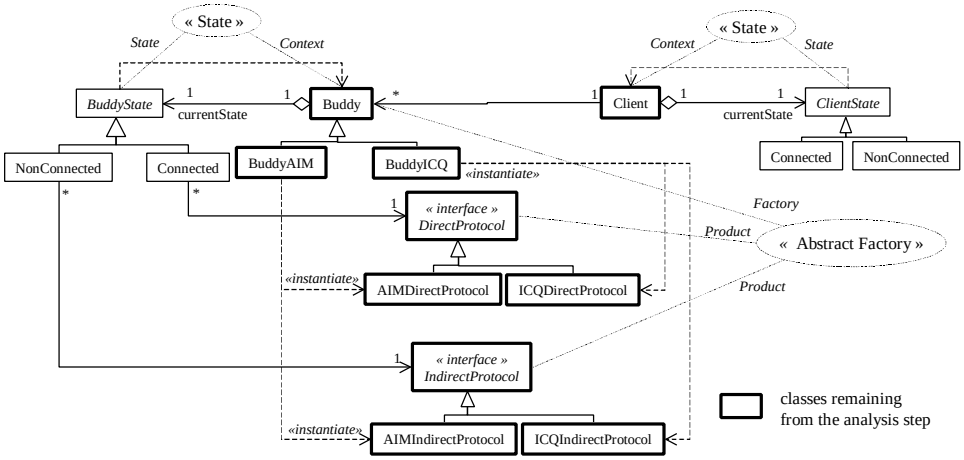

Figure 2 - An Instant-Messaging client (a possible "final" design)

A "functional" implementation of this state machine can consist of coding the machine as a single class. For example, the body of each state-dependent method is a large conditional statement. Such methods (for example send()) check the different values associated with the current state, and then use different processing for each of these values. This is illustrated in figure 4. There are at least two arguments against using this type of implementation. First, it tends to make the code unclear, difficult to read, and hard to maintain or reuse. Second, it is difficult to extend: in this case, adding a new state imposes adding a new case in every state-dependent method, i.e. changes are required in the code of multiple methods. Hence, the risk of committing an error increases.

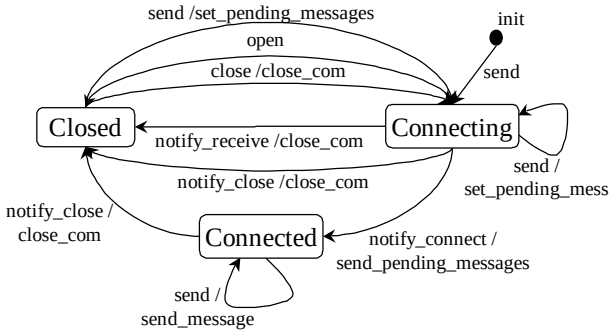

Figure 3 - State machine for Mailer class

Of course, this implementation also has to be tested for correctness of each associated method. Once the instance of the class is in the right state, each method is tested for this state and for each existing transition from this state. So, the number of statements to test this class is of the order of the average number of statements needed to put the class into a given state, multiplied by the average number outgoing transitions, plus the number of statements needed to test its behavior.

Now, let us use the State pattern [Gamma95] to implement the state machine. The architecture is shown in Figure 5). It contains the `Mailer` class and the ***`MailerState`*** interface. The latter collects every state-dependent method, and a concrete state class for every state. In the `Mailer` class, all state-dependent methods are now delegated to the `currentState` object, which can be bound dynamically to any of the ***`MailerState`*** class' children. Hence, we do not need to control the state value anymore. The right processing for a method is done thanks to dynamic binding. This makes the `Mailer` class much clearer, and what is even better, completely independent from its associated state machine. To change the state machine, we do not have to change anything in this class. The two issues with the "classical" solution are solved. The code of every method is now very small (each method computes only one operation), and thus it is more readable. Also, the architecture makes it easier to change the state machine. For example, adding a new state would consist of adding a new concrete class for this state.

```
MAILER
private final int closed = 1;
private final int connecting = 2;
private final int connected = 3;
private int currentState;
private *string pendingQueue
public Mailer(){currentState = closed;}
public void send(string mess){
  switch(currentState){
    case closed :
      print("Trying to send
          while closed : connecting");
      set_pending_message(mess);
      currentState := connecting);
    case connecting :
      print("Trying to send");
      set_pending_message(mess);
    case connected :
      print(« Effective send »); }}
…
```

Figure 4 - state machine implemented in a single class – equivalent to a "functional" design

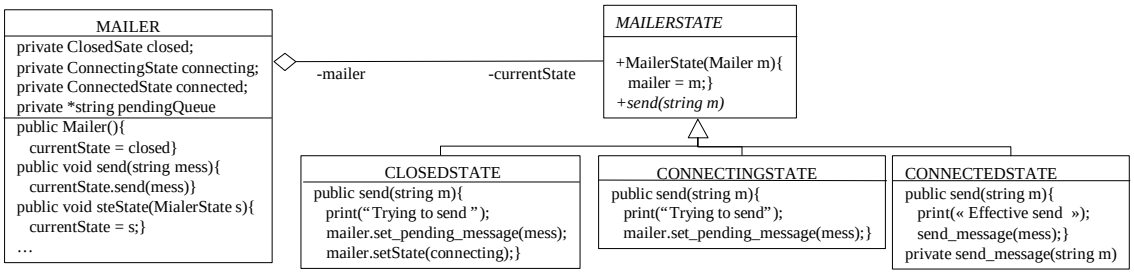

Figure 5 - state machine implemented with the State design pattern

Several elements make the pattern-refined design more testable than the "classical" design, even if it seems more complicated in terms of the number classes or in terms of the coupling between the features of the model. First, the complex hierarchical control structure of the previous implementation has disappeared. The control does not need to be tested as such since it is handled by the construction. Second, since the specific behaviors for each state are isolated in different classes, the explicit phase of putting the object in a particular state before testing it is not necessary. Indeed, the particular computation of a method for a given state can be reached directly in the class implementing this state.

However, new testing problems arise that seem specifically linked to object-oriented design and especially the micro-architecture corresponding to the design pattern based implementation. Next section presents two configurations in the design that may lead to these new testing problems.

3. Testing conflicts

In this section, we identify two main types of OO testing weaknesses found in pattern-refined designs. These can be compounded further by inheritance and polymorphism.

3.1. Weaknesses

Consider again Figure 2. A first look at this architecture reveals that many classes are strongly process inter-dependent. All the children classes are strongly linked to their parent classes, and `Client` and `ClientState` both depend on each other. This type of architecture has considerable potential for faulty behavior. For example, `Buddy` may depend on `AIMDirectProtocol` via several paths. If such usage is undesired, it has to be either tested for, or avoided by constrained construction. These potential problems have to be recognized in order to estimate the verification and validation effort. The two potential sources of problems:

- When a method m' in class Client uses a method m of class Connected, the class Connected may call back Client to process m. That means that the class Client might use itself when it uses Connected to process part of its work.
- When a class of Buddy uses ICQDirectProtocol, it might do so in two different ways: directly by declaring an instance of class ICQDirectProtocol, or through a use of Connected which uses DirectProtocol which can then be instantiated by ICQDirectProtocol.

In this context we define the potential and real testing conflicts.

Potential testing conflict. *We call a potential testing conflict, a configuration on a class diagram for which two classes, by transitivity of associations, aggregation, composition, or usage dependencies, may interact through different paths, or for which a class may use itself. It is called a potential conflict since the class diagram is only an abstract view of the software. Indeed, the conflicts detected at the design level can disappear or can worsen when the design evolves and is implemented.*

Real testing conflict. *A real testing conflict concerns interactions between objects. Some of them can be detected at the design level from Object diagrams, or sequence diagrams, but, since those diagrams can offer only a partial view of the system, and are likely to change, they cannot be used to detect all real conflicts in the system. Actual real testing conflicts can all be detected only when the source code is available: a conflict occurs when two* objects *interact, indirectly, through different objects.*

3.2. Concurrent usage testing conflicts

A class C in a system can have a usage relationship with another class E in two ways. A conflict due to this two-fold dependency appears, and testing must ensure that the internal state of the provider objects of class E cannot be corrupted by this interaction with objects from class C. Again, if these dependencies traverse inheritance trees, the conflict can become untestable. This first type of conflict is called in this paper, the *concurrent usage conflict.*

In figure 6, class C can use class E by transitive usage relationship through a set of D classes, or through a set of D' classes. For example, let C use only one class D, and one class D'. In that case, if D uses E to answer a request

from C, C has a transitive use relationship with E. In the same way, if D' uses E to answer a request from C, C has a transitive use relationship with E. When C is a client of E in both ways, it can imply a conflict, because C can change E's state using one path, and use E in another path. E's states have to be always consistent with that of C.

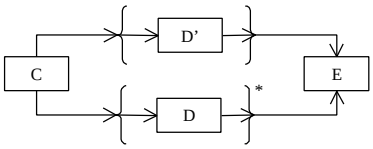

Figure 6 - Concurrent Use Relationship Example

When concurrent use exists, we must process all transitive use relationships between classes C and E to check that the provider's state is always consistent for the client.

3.3. Self usage conflict

The second type of conflict we deal with in this paper corresponds to the occurrence of a cycle of dependencies in the design. The idea here is that a class can use itself by transitive usage relationships (figure 7). For example, if class C uses D to that then uses C to provide C, class C uses itself to answer a request to itself. This is called *self-usage relationship.*

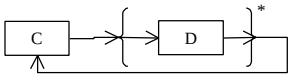

Figure 7 - Self-Usage example

At the code level, if an instance of the class uses itself, there is a real self-use conflict. Testing this conflict means testing for the absence of an infinite loop of method calls, and for the consistency of the conflict source class before and after the call to a method that makes the object call itself.

3.4. Inheritance complexity

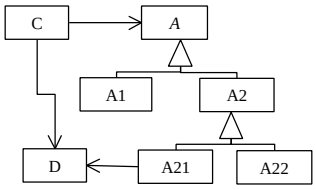

Figure 8 - Concurrent usage through an inheritance hierarchy

The complexity due to inheritance appears when the transitive (concurrent or self-use) relationship goes through several inheritance hierarchies. In figure 8 there is a potential concurrent use relationship between C and D. But the conflict is more complex when C uses an instance of class A or A2 or A21 and those three classes have relationships amongst themselves. In that case, each of the three potential use by C (A or A2 or A21) has to be tested for concurrent use conflict. For each we have to test the relationships between the classes in the inheritance hierarchy. If we constrain the design, and make it more precise, we can reduce the complexity of the conflict. Indeed, if classes A and A2 are interface classes, we can ensure that C can only use A21 or A22: the area of the conflict in class D is thus reduced to class A21.

4. Testability of Design Patterns

In this section, we show how solving testing conflicts at a micro-level. This would allow reducing the complexity of global system testing (see section 2). We illustrate the proposed testability analysis on two micro-architectures, namely instantiations of design patterns Abstract Factory and State [Gamma95]. The results are useful since they underline the difficult elements of the designs, where misleading interpretations may occur leading to a very difficult to test implementation. The second interesting result is that informal advice and simple constraints can be expressed at the design level to reduce conflicts in the implementation.

4.1. Abstract Factory

Figure 9 shows an application of the Abstract Factory Design Pattern taken from [Gamma95]. A potential concurrent usage conflict appears between the client class and each of the concrete product classes: the client class can use a concrete product directly or through the WidgetFactory inheritance hierarchy.

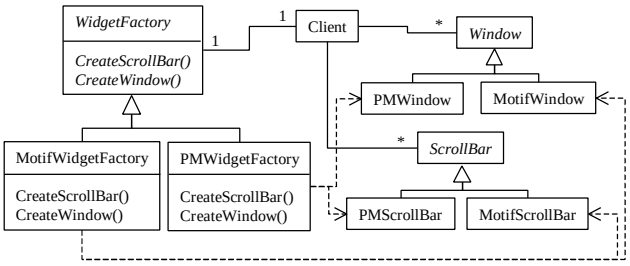

Figure 9 – An application of the Abstract Factory Design Pattern

An informal advice, on how to improve the testability of the class diagram shown figure 9, is to use interfaces for the abstract classes as much as possible: this avoids links from children to parents. If only leaf classes in the inheritance hierarchy (classes that have no descendants) are concrete classes, the complexity of the conflict going through this hierarchy is reduced since there are no interactions between classes in the hierarchy.

To test the application of the Abstract Factory pattern, we must check if the delegation from the client to the factory creates all objects and does not do anything else. If the design pattern application is well implemented, the client class uses creation methods of concrete products

through the WidgetFactory inheritance hierarchy and uses other methods directly calling the concrete products objects. In that case there is no conflict at all since the concurrent paths between the Client and the Products are used at different moments.

Typical value for testing conflicts when using an Abstract Factory pattern (example from [Gamma95], figure 9):

- 4 potential conflicts
- with all abstract classes converted to interfaces: 4 conflicts less complex
- if delegation is well implemented: 0 conflict

Concerning the third point, the solution would be to express implementation constraints on the design to specify clearly the delegation. For example, the association from MotifWidgetFactory to MotifWindow could be labeled with a «create» UML stereotype. This informs the developer that MotifWidgetFactory uses only creation methods of class MotifWindow.

4.2. State

There are multiple cycles around the context class (figure 10) that create a self usage potential conflict.

As for the Abstract Factory pattern, an informal advice to improve the testability of the class diagram shown figure 10 is to use as many interfaces as possible.

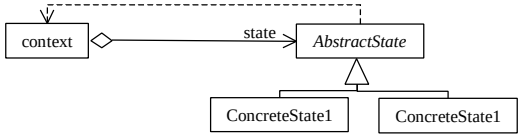

Figure 10 – An application of the State Design Pattern

To test the application of the State pattern, we have to test that concrete states use the context class to change its state attribute, and do not change any other attribute. If this is well implemented, there is no conflict.

Typical value (example from [Gamma95]):

- 2 potential self use conflicts
- with all abstract classes converted to pure interfaces: 2 conflicts less complex
- if delegation is well implemented: 0 conflict

5. Conclusion

This paper discusses two configurations of an OO design 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. Using two of them, the State and the Abstract Factory, we illustrated how testing risks might be avoided. The two risk mitigation techniques we used are design information enhancement, and design refinement constraining. 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 such 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 [Sunyé00]. 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.

Acknowledgment: Many thanks to Mladen A. Vouk for his helpful comments and suggestions for improving this paper.

Bibliography

[Binder94] R. V. Binder, "Design for Testability in Object-Oriented System", Communications of the ACM, Vol. 37, No. 9, pp87-101, September 1994.

[Binder99] R. V. Binder, "Testing Object-Oriented Systems: Models, Patterns, and Tools", Addison-Wesley, October 1999. ISBN 0-201-80938-9.

[Briand01] L. Briand, Y. Labiche, "a UML-based approach to System Testing", Technical report, 2001, Carleton University, Ottawa.

[Dsouza98] D'Souza D. F., Wills A. C., "Object, Components and Frameworks with UML, The Catalysis Approach", Addison-Wesley Object Technology Series, Addison-Wesley, 1998. ISBN 0-201-31012-0.

[Gamma95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley Professional Computing Series, Addison-Wesley, 1995. ISBN 0-201-63361-2.

[Jézéquel99] J.-M. Jézéquel, M. Train and C. Mingins, "Design Patterns and Contracts", Addison-Wesley, 0ctober 1999. ISBN 0-201-30959-9.

[Sunyé00] G. Sunyé, A. Le Guennec, and J.-M. Jézéquel, "Design pattern application in UML", in proceedings, no. 1850, pp. 44-62, Lecture Notes in Computer Science, Springer Verlag, June 2000.

[Voas91] J. M. Voas, K. Miller, "Software Testability: The New Verification", IEEE Software, Vol. 12, No. 3, pp. 17-28, May 1995.

[Voas96] J. M. Voas, "Object-Oriented Software Testability", 3rd international conference on achieving quality in software, January 1996.