Testing Object-oriented Software Systems

Harry M. Sneed
Anecon GmbH., Vienna
Alserstrasse 4
A-1090 Vienna, Austria
#43-1-40958900
Harry.Sneed@anecon.com

ABSTRACT

Object-oriented Software Systems present a particular challenge to the software testing community. This review of the problem points out the particular aspects of object-oriented systems which makes it costly to test them. The flexibility and reusability of such systems is described from the negative side which implies that there are many ways to use them and all of these ways need to be tested. The solution to this challenge lies in automation. The review emphasizes the role of test automation in achieving adequate test coverage both at the unit and the component level. The system testing level is independent of the underlying programming technology.

Keywords

Object-oriented software, error sources, unit-testing, integration-testing, test coverage, test automation.

1. The challenge of testing object -oriented software

Back in the early 1990's when object-oriented development was getting started there were already critics pointing to the difficulties involved in testing. There is always a negative side of every supposed positive development and the negative side of object-oriented seemed to be the test. The first of the Cassandra voices was that of Gail Kaiser and William Perry who claimed to have discovered a major flaw in the general attitude about object-oriented Software, namely that proven classes can be reused as super classes without retesting the inherited code. Kaiser and Perry demonstrated that an object-oriented class hierarchy has to be retested for every context in which it is reused. They did not negate Barbara Liskow's principle of substitution but they qualified it. Substitution is theoretically correct, but practically expensive, since little differences in the environment can cause big side effects. [1]

Another early skeptic was James Martin, the guru of data – driven software development who pointed out, that reusing classes from class libraries was like eating spaghetti. If you wanted to pull out one strip all of the other strips come along too. They are all intertwined with one another. So it is not possible to select and test individual classes. If you want to reuse and test one, you have to test them all. [2]

David Taylor, one of the early proponents of object-oriented development, one who even suggested restructuring business processes from an object–oriented point of view, admitted that for most programmers, the amount of testing required for object-oriented software may seem like a major intrusion on what should be a quick, intuitive process of class development. He was implying that the greater part of the development effort has to be devoted to testing. [3]

He was seconded in this by none other than Boris Beizer, the dean of Software testing, who wrote in an article in the American Programmer, that it would cost a lot more to test 00-software than to test conventional procedural software – perhaps 3 or 4 times as much. [4]

One might write his comments off as an attempt to promote the need for testing, but another proponent of object – orientation and the author of the first book on testing object-oriented systems, Stan Siegel, admitted that while many advocates of object technology are suggesting that testing needs will be lower for 00-systems, in reality this will not be true. Object-oriented systems require much more testing effort. [5] The reasons for this are documented in his book, the first book on this subject. [6]

Since then there has been no lack of testing experts all pointing to the fact that 00-software requires more testing, foremost among them Robert Binder, who authored an 1800 page book on the testing of object–oriented systems in 1999. Binder collected a long list of techniques for testing object–oriented systems at the unit, component and system level, but none of these techniques actually reduce the effort required. If anything, they only increase it. To decrease effort without increasing the risk of errors, Binder suggests using one or more of the many automated testing tools that are available in the market. According to Binder, automation is the key to economically testing object–oriented systems. [7]

2. Cost Drivers in Testing Object – Oriented Software

If done correctly the testing of conventional, procedural software is also not cheap. The goal of C1 test coverage, i.e. traversing every branch in the control graph, required a lot of test cases, at least as many as paths through the control graph. [8] However, procedural components are generally designed to solve one particular problem. Therefore, they have a limited number of inputs and outputs, only as many as required to solve that particular problem. If one wants to solve many problems of the same type, one would write as many programs as there are problem types. The testing is done one instance after the other. The effort to test any one of these programs is limited by the scope of that program type. In testing procedural programs one is always testing only one problem at a time.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ECOOP '2010 Maribor, Slovenia EU
Copyright 2010 ACM 978-1-4503-0538-9/10/06 ...$10.00.

In object–oriented systems one is attempting to find a general solution which will fit a whole class of problems of a specific type. Therefore all of the paths and all of the inputs and data states of all problem variants have to be tested. Not only that, but any good object–oriented designer will attempt to generalize the solution to the point that it will not only cover the currently known problem types, but also all potential problem of the class that may come up in the future. By means of abstraction, and generalization and using virtual interfaces, it is possible to achieve this without increasing the size of the code. However, the functionality does increase. It is as great as if the code were duplicated for every variant. Thus object-orientation succeeds in decreasing the code but not the functionality and it is the functionality which has to be tested.

Therefore, there has to be a test for every sub problem. The test is being made at the object code or byte code level. For every variant different object codes are being generated from the same source code at compile time and for every object type multiple instances are being created at run time, all of which represent a different state. Each problem variant has not only its own code version, with all of the paths through that version, but also all of the data states which may came up in that variant. In the end we have to test much more than what we coded, as Beizer points out, 3 to 4 times more depending on the number of problem variants. [9]

Particular drivers of test costs are exactly those features which distinguish object-oriented software from procedural, problem specific software. These are;

- encapsulation
- generalization
- association and
- polymorphy [10].

Encapsulation restricts access to the data states of a class. A data state is represented by an object instance. In testing it is necessary to be able to manipulate data states and to check their content. In procedural systems it is possible to gain access to each and every field in the storage of a program since they are all kept in a single static storage space. These fields can be set, altered and checked indiscriminately.

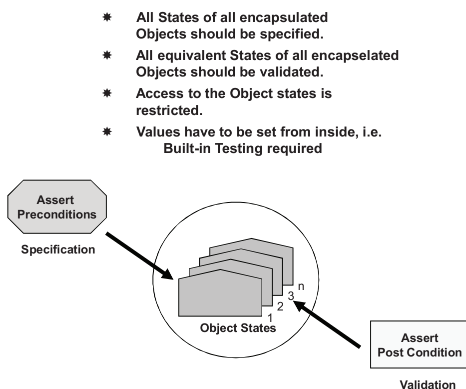

Figure 1: Consequences of Encapsulation

In object-oriented systems it is not so easy. To get to a particular data field, it is necessary to go through the constructor and access methods of the class. That is the reason for the emphasis on built-in tests when testing classes. The test has to be part of that particular class under test. Since there is a different test for every class, that increases the test effort.

Generalization increases test effort by requiring us to test every context in which a super class can be used. Additional sub classes can be added at will, but one can never be sure that the same super class really fits to them. As pointed out by Perry and Kaiser each new branch of an inheritance tree has to be tested on its own merits. The deeper the inheritance tree is, the more branches there are to test, and each additional branch of the tree may have side effects on the other branches, so these too have to be retested.

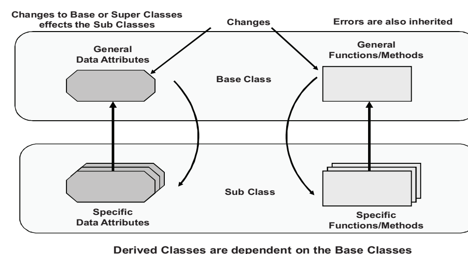

Figure 2: Effect of Inheritance on Testing

Association allows for any object to be associated with any other object in the system. This is accomplished by invoking a foreign method in another class hierarchy. Each such association has to be tested. This would not be so bad if the associations were limited. In practice they are not. Classes invoke methods in a neighboring class which invoke methods in their neighboring class and so on. What happens is an association explosion which in the end may affect every class hierarchy in the system. As James Martin put it, it is like eating spaghetti. You pull out one strand and you get the whole bowl. This makes it difficult to test individual classes because one has to simulate all of their associated classes.

Polymorphism makes it possible to select methods for execution at run time. This makes the code much more flexible and general, but it is a nightmare for testing. Testing a polymorphic method invocation means testing every potential target method. It is true that this problem also exits in procedural software in the form of dynamic links but ii is just as much as a problem there. The fact is that flexible, dynamically variable software causes a greater testing effort, since every possible variation of the software should be tested. For that reason such flexible constructs should be avoided in safety critical applications. They only increase the risks. In the words of Edsger Dikstra polymorphism can be considered harmful. It is the GOTO of object-orientation.

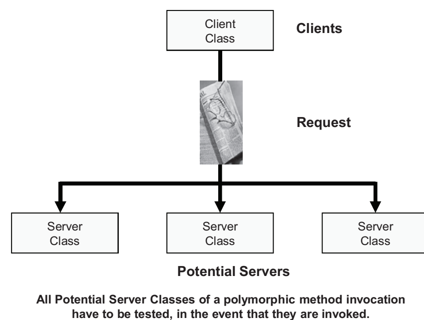

Figure 3: Testing of Polymorphism

One can summarize that the greater the variability and flexibility of a piece of software, the more the usage possibilities are, then the more there is to test. Everything has its price.

3. Testing object–oriented classes

This fact is demonstrated by the different approaches to testing classes. Classes contain methods and object instances. Each object instance represents another state. The methods may also have multiple paths through them, for instance in the case of decision statements and loops. In testing procedural units, i.e. modules, one was aiming at traversing every path through the module. In testing object-oriented units, i.e. classes, one also has to deal with state. Not only that, but since individual methods can be invoked from outside, one must test every potential chain of methods, from entry to exit.

Binder has pointed out four approaches to class testing

- non modal
- uni modal
- quasi modal
- modal [11].

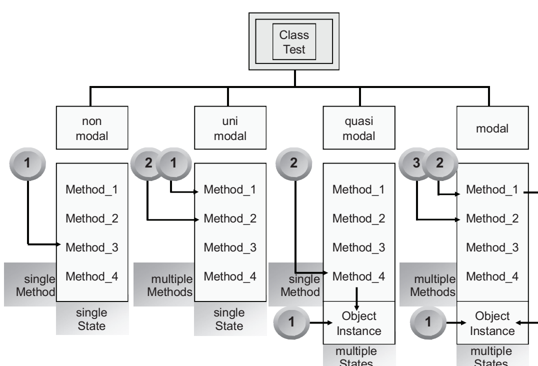

Figure 4: Approaches to Class Testing

Non modal means testing every single method with a single object instance. Uni modal means testing every combination of methods with a single object instance. Quasi modal implies testing individual methods with multiple object instances and modal implies testing every combination of methods with multiple object instances, e.g. all possible object states. On top of this comes the conventional challenge of testing every path through a method. The result of combining paths, with methods, with states leads to a combinational explosion of potential test cases. The only way to deal with this is by automatically generating the test cases out of the code and joining these test cases with those derived from the class specification. Without the support of automated class test beds there is no way to test complex classes. Tools like JUnit, CPPTest and NetUnit can fulfill this need if used correctly..

4. Testing the Integration of object–oriented Units

Classes are the nodes of a class hierarchy. Within the class hierarchy classes are connected via inheritance and association. Inheritance gives lower level classes access to the methods and attributes of the classes above them. Association allows a class to use methods in neighboring classes. Both of these connection types needs to be tested for all occurrences. Siegel speaks of

- vertical and
- horizontal

integration testing [12].

Vertical testing is the testing of the inheritance connections. Horizontal testing means testing the associations. At the class hierarchy level, integration testing means validating each generalization and each association among classes in a single hierarchy.

Complex components may, however, contain several class hierarchies which are dependent on one another. This dependency is expressed in the associations between methods in separate class hierarchies. If this is the case, then all of these dependencies need to be tested for all of the object states that may exit when the associations occur. Here too, test cases are to be created for each potential interaction type. Both control flow and data flow have to be considered.

The greatest challenge in integration testing is that of testing separate components interacting with one another. There are several ways for components to interact, e.g.

- they may share a common database
- they may call methods in the other component or
- they may exchange messages and responses [13].

The integration test of components is based on the interface descriptions. As such it is in effect an interface test. The test cases are extracted from the interface specification. Whether it be in CORBA-IDL, XML, WSDL or SQL. The goal is to test all potential combinations of arguments [14]. Since there may be too many arguments to justify the costs, it may be necessary to select certain representative combinations. The first approach is automatable. The second requires some form of human interaction to select combinations of states which are representative of the

others. It is here where techniques like boundary analysis, equivalence classes and cause and effect analysis come to play. Interface states, i.e. messages, have to be generated and the responses validated. This can be done automatically by integration testing tools which process the interface descriptions and extend them by data value domains. Thus, component integration testing is a mixture of manual and automated techniques aimed at demonstrating the correct behavior of components interacting with one another [15].

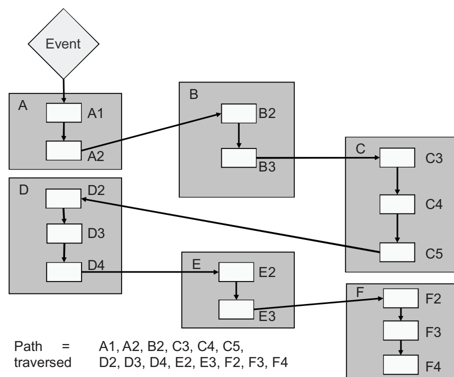

Figure 5: Integration of Distributed Objects

This same type of test can be extended to test the interaction between separate subsystems. Here too the test is based on the interface definitions. At this point it is no longer relevant whether the subsystems are object–oriented or not. As a rule new object–oriented components and subsystems will be integrated together with legacy components and subsystems. The only prerequisite to this integration is the proper wrapping of the legacy components. Then it no longer matters how the components and subsystems are implemented. It only matters that they behave as expected. To prove that the expected behavior has to be formally specified and then validated by automated tools interpreting the formal specification.

The main prerequisite to integration testing is a formal interface specification language, which also includes the value domains of the arguments and results. These can be ranges, equivalence classes or conditional relationships. Such a language has been implemented by the author for the DataTest integration test tool [16].

5. The system test for object–oriented systems

At the system level it should not be necessary to distinguish between procedural and object–oriented systems. The test will be the same in both cases. One is testing the functionality as a whole. What can be different are the error types that occur. In testing an object–oriented system one should not get errors like data exception or storage overflow, since these errors are characteristic of procedural software. On the other hand, one will get errors like class cast exception or null object reference, which are typical of object–oriented software. The system testers should be trained in recognizing and tracking such object related errors. Otherwise, one should not distinguish between the test of an object–oriented system and that of a procedural one. In many cases, a complex system will contain elements of both [17].

6. Test-driven Development

The challenge of testing object-oriented software has given rise to the test-driven approach. According to this approach, the developer should first create a test framework for every class or group of classes he wants to develop. This framework includes a test class which initializes requests and invokes the methods to be tested. For every method which can be invoked from outside, a series of test cases is specified. Thus even before a method has even been coded, there already exist tests to test it. Then the developer proceeds to code one method at a time and to test it immediately to ensure that it fulfills its mission according to the test cases. This prudent approach is an essential element of extreme programming [18]

For integration testing, the approach is carried over to the testing of interfaces. The specification of an interface, such as a web service request, is coupled with the test cases to test that interface. As a matter of fact, the test cases are built into the interface definition. This is to ensure that the interface can be both generated and validated. At both the unit and the component level the test has predominance over the implementation.

7. Model-driven Testing

Finally we come to the latest approach to testing object-oriented components. That is model-driven testing. In this approach the test is derived from the object model, in particular from the UML diagrams. Test cases can be automatically extracted from the use case diagrams, activity diagrams, state diagrams and sequence diagrams to test all features prescribed in those diagrams. For instance from the state diagrams test cases are derived to test every state transition, and from sequence diagrams test cases are generated to test every object interaction. In principle, this approach is not really new nor is it limited to object-oriented software. Earlier tests were generated from the procedural design documentation.

The drawback of model-driven testing is that the test is only as good as the model and we know from practical experience that models are seldom complete and most often inconsistent. This again is an approach which is theoretically appealing but does not hold up in practice. The test-driven development approach is more likely to be used.

8. Conclusion

The conclusion of this short expose is that object–orientation is foremost a problem at the unit and integration testing levels. There it leads to an increased testing effort due to the increase in possible usages and to the greater number of potential interactions. The increased flexibility and usability of the software under test makes it all the more important to automate the test since only an automated test can cover the code in combination

with the many object states. By automating the test, emphasis is shifted from the manual creation of test cases for selected paths to the manual creation of test scripts from which test cases for each paths and each set of relevant object states can be generated. In testing object – oriented systems, the test is moved up to a higher level of abstraction, where test automation is absolutely necessary.

9. References:

[1] Perry D., Kaiser G.: ”Adequate Testing and Object–oriented Programming”, Journal of 00-Programming, Vol. 2, No. 5, Jan. 1990, p. 13

[2] Martin, J., Odell, J.: Object–oriented Analysis & Design, PTR Prentice Hall, Englewood Cliffs, 1992, p. 29

[3] Taylor D.: ”Quality First Program for Object Technology”, in Object Magazine, June 11992, p. 17

[4] Beizer , B.: ”Testing Technology – The Growing Gap”, in American Programmer, Vol. 7, No.4, April 1994, p. 9

[5] Siegel, S.: ”Strategies for Testing Object–oriented Software”, CompuServe CASE Forum Library, Sept. 1992

[6] Siegel, S.: Object–oriented Software Testing – A Hierarchical Approach, John Wiley & Sons, New York, 1996, p. 16.

[7] Binder, R.: Testing object-oriented Systems – Models, Patterns and Tools, Addison – Wesley Longman, Reading, MA. 1999, p. 801

[8] Ould, M., Unwim, C.: ” Testing in Software Development”, Cambridge University Press, Cambridge, 1986, p.8

[9] Binder, R.: ”Design for Testability in Object–oriented Systems”, Comm. of ACM, Vol. 37, No. 9, Sept 1994, p. 112

[10] Dlugosz, J.: ”The Dark Side of OOP”, in American Programmer, Vol. 7, No. 10, Oct. 1994, p. 12

[11] Binder, R.: ”Class Modelity and Testing”, Object Magazine, Vol. 7, No. 2, Feb. 1997, p. 48

[12] Jorgensen, P., Erickson, C.: ” Object–oriented Integration Testing”, Comm. of ACM, Vol. 37, No. 9, Sept 1994, p. 132

[13] Orfali, R., Harkey, D., Edwards, J.: The Essential distributed Objects Survival Guide, John Wiley & Sons, New York, 1996, p.221

[14] Nguyen, H.Q.: ” Testing Web-based Applications”, Software Testing & Quality Engineering, Vol. 2, No. 3, 2000, p. 23

[15] Gros, H-G.: Component–based Software Testing with UML, Springer Pub., Heidelberg, 2009, p. 11

[16] Sneed, H., Winter, M. : Testing Object-oriented Software, Hanser Verlag, Munich-Vienna, 2001, p. 217

[17] Beizer, B.: Black–Box Testing – Techniques for Functional Testing of Software Systems, John Wiley & Sons, New York, 1999, p. 11

[18] Beck, K.: Test-Driven Development, Addison Wesley Publishing, Boston, 2003, p. 24