

Testing Object Oriented Software

Mauro Pezzè
Università degli Studi di Milano Bicocca
Dip. Informatica, Sistemistica e Comunicazione
Via Bicocca degli Arcimboldi, 8
20126 Milano (Italy)
pezze@disco.unimib.it

Michal Young
University of Oregon
Dept. of Computer Science
1202, Deschutes Hall
Eugene, OR 97403 (USA)
michal@cs.uoregon.edu

1. Test and Analysis in the context of OO software

Object oriented software requires reconsidering and adapting approaches to software test and analysis. Characteristics of object oriented software that impact test and analysis include state dependent behavior, encapsulation, inheritance, polymorphism and dynamic binding, abstract classes, exception handling, and concurrency.

While test designers should be prepared to adopt new testing approaches for object-oriented software, they cannot ignore traditional technology and methodologies. An object oriented design approach mainly affects detailed design and code, but planning, requirements analysis, architectural design, deployment and maintenance are largely independent of the use of a specific design approach and can be and should be addressed with traditional methods and technology.

The best approach to testing object oriented software depends on many factors: the application-under-test, the development approach, the organization of the development and quality assurance teams, the criticality of the application, the development environment and the implementation language(s), the use of design and language features, project timing and resource constraints. Nonetheless, we can outline a general approach that works in stages from independent consideration of classes and their features to consideration of their interactions. A coherent strategy would include three main phases: intra-class, inter-class, and system and acceptance testing.

Intra-class testing deals with classes in isolation and includes testing of abstract classes, selection of test cases from the ancestors' test suite, and testing state-dependent behavior of the class under test. *Inter-class* testing applies to clusters of classes to be tested incrementally, considering class interactions, polymorphic calls, and exception handling. *System and Acceptance* testing considers the software as a whole independently from its internal structure and re-

lies on traditional system and acceptance testing techniques.

2. State-dependent behavior and encapsulation

Since the state of an object is implicitly part of the input and output of methods, we need a way to systematically explore object states and transitions. This can be done using a state machine model, which can be derived from module specifications. State machines can be traversed with different criteria for deriving state based test cases: history-insensitive criteria produce relatively small sets of test cases, and are thus well suited for testing large applications with stringent cost constraints; history-sensitive criteria produce more thorough test suites that contain more test cases, and thus apply well to small applications with stringent quality requirements.

When moving from intra- to inter-class testing, we must combine different state machines. Approaches that consider all combinations of possible interactions are usually impractical. Practical criteria can be derived from combinations of random selection of interactions with explicit testing of significant interaction scenarios that have been identified in design and analysis, e.g., in the form of UML interaction diagrams.

Structural testing criteria must also take into account object state. Since the state of a class is comprised of the set of values of the instance variables of the class, the number of states can be enormous. We can radically reduce the size of test sets while remaining sensitive enough to catch many common oversights by modeling the points at which the variables change value. This is the same intuition behind data flow testing, although it requires some extension to cover sequences in which one method defines (sets) the variable, and another uses the variable.

Intra- and inter-class testing require suitable scaffolding to exercise the classes under test (drivers and stubs) and to inspect the test results (oracles). Constructing stubs and

drivers for object-oriented software is essentially similar to constructing stubs for procedural programs. Oracles, however, can be more difficult to construct, owing to encapsulation of object state. Oracles need to check the validity of both output and state. Unfortunately the state of objects may not be directly accessible. Oracles built by violating encapsulation may result in differences in behavior between what is tested and what is used. An effective alternative to violating encapsulation is to provide a way of determining whether two objects are equivalent, i.e., they represent the same abstract value.

3. Inheritance, polymorphism, and dynamic binding

Inheritance does not introduce new classes of faults, but it provides an opportunity for optimization by re-using test cases and even test executions. Subclasses share methods with ancestors. Identifying which methods do not need to be retested and which test cases can be reused may significantly reduce testing effort.

Limited use of polymorphism and dynamic binding can be easily addressed by unfolding polymorphic calls, i.e., replacing each polymorphic call with all methods that can be dynamically bound to that call. Complete unfolding becomes impractical when each reference to an object could be bound to instances of several sub-classes. We can select an effective subset of combinations using a combinatorial testing approach.

Generics are designed to be instantiated with many different parameter types. We can test only instantiations, not the generic class itself, and the problem is that we may not know in advance all the different ways a generic class might be instantiated.

Programs in modern object-oriented languages use exceptions to separate handling of error cases from the primary program logic, thereby simplifying normal control flow. The price of separating exception handling from the primary control flow logic is introduction of implicit control flows. Treating every possible exception as an explicit control flow would create an unwieldy control flow graph. It is more practical to consider exceptions separately from normal control flow in test design.

4. Further Reading

Many recent books on software testing and software engineering address object-oriented software to at least some degree. The most complete book-length account of current methods is Binder's *Testing Object Oriented systems* [1].

State-based testing has been important in the domain of communication protocol development and conformance testing; Fujiwara, von Bochmann, Amalou, and Ghedamsi

[4] is a good introduction. Thévenod-Fosse and Waeselynck describe statistical testing using statechart specifications [9]. Other approaches have been proposed by Chen and Kao [2], Hartmann et al. [6], Kung et al. [7], and Turner and Robson [10]. Structural state-base testing is discussed in detail by Martena, Orso and Pezzè [8]. An excellent paper by Doong and Frankl [3] introduces equivalent scenarios.

Harrold, McGregor, and Fitzpatrick [5] provide a detailed discussion of the use of testing histories for selecting test cases for subclasses.

References

- [1] R. V. Binder. *Testing Object-Oriented Systems - Models, Patterns, and Tools*. Addison-Wesley, 2000.
- [2] H. Y. Chen, T. H.Tse, F. T.Chan, and T. Y.Chen. In black and white: an integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 7(3):250–295, July 1998.
- [3] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, April 1994.
- [4] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.
- [5] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick. Incremental Testing of Object-oriented Class Structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 68–80, May 1992.
- [6] J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-based integration testing. In *Proceedings of the International Symposium in Software Testing and Analysis (ISSTA '00)*, pages 60–70. IEEE Computer Society Press, 2000.
- [7] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, Y.-S. Kim, and Y.-K. Song. Developing and object-oriented software testing and maintenance environment. *Communications of the ACM*, 38(10):75–86, October 1995.
- [8] V. Martena, A. Orso, and P. M. Interclass Testing of Object Oriented Software. In *Proceedings of the 8th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'03)*, December 2003.
- [9] P. Thévenod-Fosse and H. Waeselynck. Towards a statistical approach to testing object-oriented programs. In *Proceedings of The 27th Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 99–109. IEEE, June 1997.
- [10] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. In *Proceedings of the International Conference on Software Maintenance*, pages 302–310. IEEE Society Press, September 1993.