OPEN ISSUES IN TESTING OBJECT-ORIENTED SOFTWARE

Stéphane Barbey, Manuel M. Ammann and Alfred Strohmeier
Swiss Federal Institute of Technology, Lausanne, Switzerland

Published in ECSQ '94 (European Conference on Software Quality), Basel, Switzerland, October 17-20 1994.

Abstract

While the use of object-oriented development methods has increased the quality of software by leading to better, modular architectures, it does not provide correctness by itself. Thus, testing software remains an important task even in the presence of object-orientedness.

The traditional testing methods decompose software into procedures. However, the dimensions of object-oriented programming are such that this decomposition is ill-fitted for testing object-oriented programs. We address the problems of testing in the three dimensions of object-oriented software: classes, hierarchies, subsystems. The class forms a well-defined basic test unit; however its testing is impeded by encapsulation and inheritance issues. The class hierarchy concept introduces problems of undecidability due to polymorphism. Subsystems are the building blocks in integration testing.

We also study the use of traditional methods and the modifications that are needed to apply them to object-oriented software, e.g. complexity metrics.

Stéphane Barbey, Manuel M. Ammann, Alfred Strohmeier, Software Engineering Laboratory, Swiss Federal Institute of Technology, IN-Ecublens, 1015 Lausanne, Switzerland, ph. +41 21 693 52 43, fax. +41 21 693 50 79, e-mail: {stephane.barbey, manuel.ammann, alfred.strohmeier} @ di.epfl.ch.

1. Introduction

Object-oriented development is viewed by many as the silver bullet that will solve the software crisis. The confidence found amongst its practitioners is sometimes such that "object-oriented" has become a thaumaturgy. However, while the use of object-oriented development methods has increased the quality of software by leading to better, modular architectures, it does not provide correctness by itself, because, alas, "*software is created by error-prone human*" (Boehm, in (Rook, 1990)). Thus, testing software remains an important task, even in the presence of object-orientedness. Since object-oriented technology promotes reuse, it could be argued that the need for meticulously tested components is even more important.

Furthermore, contrary to some belief, traditional testing methods cannot be applied directly to object-oriented software. Some aspects of the nature of the object-oriented paradigm, such as encapsulation, inheritance and polymorphism, introduce problems that were previously not found in structured software and require adapting of the testing methods used.

In this paper, we will discuss the key issues in testing object-oriented software, most of them being still open. First, we will examine what the key points of an object-oriented architecture are in order to identify the constructions that need testing. We will then expose the issues in testing the identified constructions, classes, hierarchies, and subsystems with respect to the most important aspects of the object-oriented paradigm: encapsulation (visibility problems), inheritance (incremental testing problems) and polymorphism (undecidability problems).We will also discuss the problems of metrics for testing object-oriented software.

1.1. Testing

Testing is used to gain confidence that the implementation of a program meets its specifications. It is the process of finding programmers' errors or program faults in the behaviour or in the code of a piece of software, but it cannot prove the correctness of a program. Testing is usually accomplished in several phases: the first phase is the unit testing, which focuses on testing small building blocks of the program rather than the program in its entirety.

Two complementary strategies are applied for that purpose:

- specification-based testing to check that the behaviour of a program meets its specification (i.e. regardless of its code) and
- program-based testing to gain confidence in a program by simulating the possible flows of execution of a program (the test cases are built by examining the code to be tested).

After the unit testing has been completed, the combination of units must be tested together to check that the program itself meets its specifications. This phase is called integration testing. The remaining test phases, which are acceptance and operational tests (Rook, 1990), are not affected by the development methodology and are therefore beyond the scope of this paper.

2. The dimensions of object-orientation

In this section, we will examine what the main constituents of an object-oriented architecture are.

2.1. Objects and classes

The main difference between traditional structured software and object-oriented software is that the basic constituent of an object-oriented architecture is the object instead of the algorithm. An object is an item that represents a concept which is either abstract or depicts an entity of the real world (Booch, 1991). It is usually made up of two kinds of properties: a set of features and a set of operations. The features can be values or other objects and constitute the state of the object while the operations are the subprograms that represent its behaviour and can give information on or alter the state of the object.

A class is the template from which objects are instantiated. It encapsulates the properties of its instances and can hide the data structures and other implementation details that are not to be available outside of the class. The non-hidden properties form the interface of the class. They are usually operations only. Therefore programmers can manipulate objects only by invoking these public operations and do not have to care about the data representation of the class. This separation between the specification of a class and its implementation is very important: the same specification can lead to multiple implementations. Since classes encapsulate a complete abstraction, they are easily isolated and can be reused in many applications.

For instance, in the development of a security system the design of a class *Alert* could consist of features such as time and location where an alert is raised, and operations such as *Handle*, which implements the actions to be performed upon an alert.

2.2. Class hierarchy (derivation class)

A descendant is a class derived from one (single inheritance) or more (multiple inheritance) parent classes (its ancestors). It inherits their properties, features and operations, and can be refined by adding new properties or modifying operations (overriding them) to give them a new implementation. In some inheritance schemes, it is even possible for a descendant to decide not to inherit all the operations of its parent. The set of all classes derived directly or indirectly from a given class (the root class) forms a hierarchy.

For example (see figure 1), we could create a hierarchy of alerts, which is based on their priorities. The operation *Handle* could have a different implementation for each class in the hierarchy to specify actions specific to the priority of the alert. This alert-handling system example is inspired from (Barnes et al., 1993).

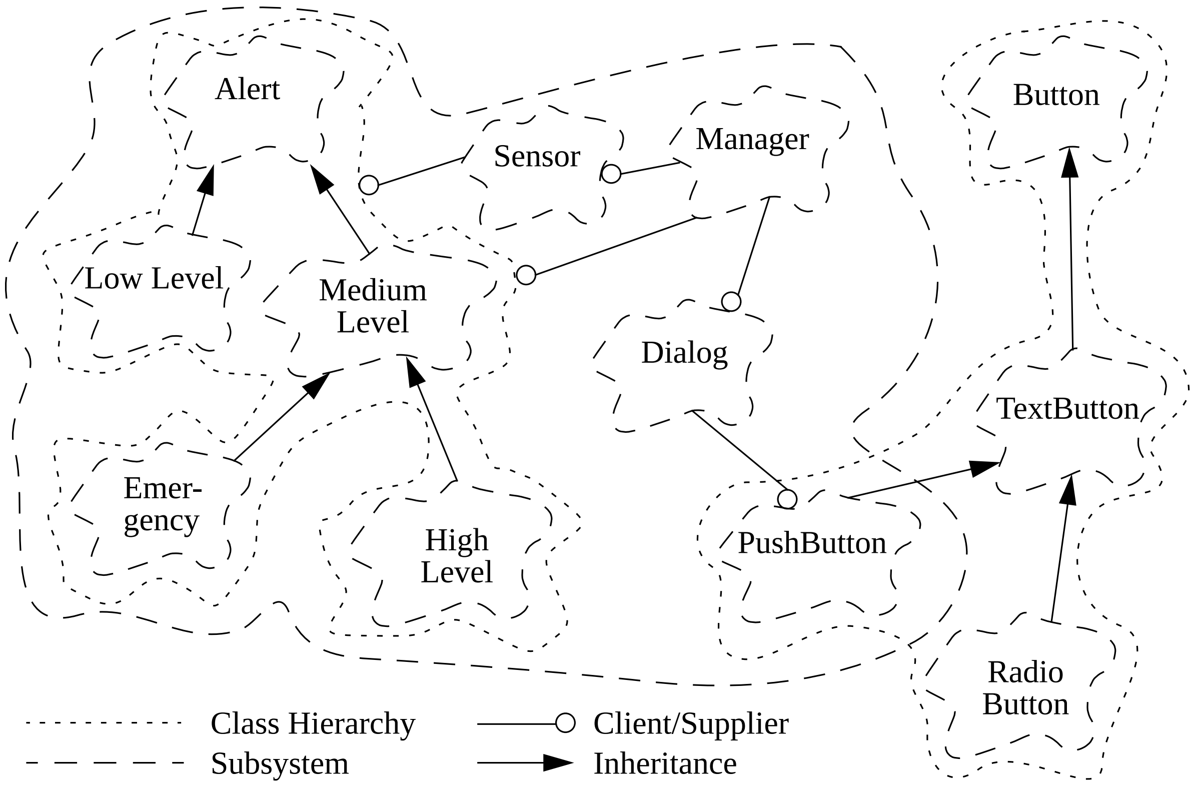

Figure 1. the dimensions of object-oriented programming

In the inheritance scheme in which all properties are inherited, each instance of a class belonging to a hierarchy owns a common set of properties which are the properties defined for the root class. Hence, it is possible to work with objects knowing only the hierarchy to which their class belongs, and not the class itself. In programming languages this possibility is often realized by a mechanism called polymorphism, which makes it possible for a name, i.e. a variable or a reference, to denote instances of various classes in the same hierarchy. Since there are different possible implementations of an operation within a hierarchy, it is impossible for the compiler to choose which implementation must be used when invoking an operation on an object denoted by such a name. This choice is postponed until run-time, when it will be made according to the class of the object denoted by the name. This mechanism is called dynamic binding.

2.3. Subsystems

A system is a collection of classes and objects used to implement an application. Since systems can be very big, they are usually decomposed into subsystems.

Subsystems are logical collections of classes and objects that collaborate to provide a set of related services. Each subsystem defines an interface that the rest of the system will use to invoke its services. Subsystems of a general nature can have a high degree of reusability

For instance (figure 1), we could imagine an alert-handling subsystem made up of an instance of *Sensor*, which detects the apparition of *Alerts* of any priority[1] and informs an instance of *Manager*, which logs the alert and brings up to the display of the action officer an instance of a *Dialog* box consisting of several instances of *Pushbuttons*[2] representing the different actions

1. The whole Alert hierarchy is in the subsystem.
2. The rest of the Button hierarchy is not in the subsystem.

that can be undertaken, e.g. ringing the alarm or handling the alert, in which case the instance of *Manager* invokes the operation *Handle* defined for the occurred alert regardless of its implementation. The interface of this subsystem will consist of the public operations of the class *Manager*, which can activate and deactivate the subsystem, and possibly proceed to an audit of the logged alerts.

Such a subsystem could then be reused by bigger systems, for instance an air traffic control system or the management of a nuclear power plant.

3. Testing classes

Most of the work in testing object-oriented software has been concerned with testing classes. It is not possible to consider a subprogram as the basic unit of testing anymore. For object-oriented approaches, the basic unit of organization is the class construct. Although the operations of a class could admittedly be tested individually, it is impossible to reduce the testing of a class to the independent testing of its operations, i.e. the tests cannot be designed, coded, and performed as if the operations were “free-floating” subprograms instead of being all related to an object:

- An operation is not a testable component, i.e. it can only be tested through an instance of a class.
- It is impossible, unlike in procedure-oriented (traditional) testing, to build either a bottom-up or a top-down testing strategy based on a sequence of invocations since there is no sequential order in which the operations of a class can be invoked unless inspired by common sense (for instance invoking the operation that creates an object before invoking any other operation of the object).
- Every object carries a state. The context in which an operation is executed is not only defined by its possible parameters, but mainly by the values of the features of the object by which it is invoked (i.e. its state). Furthermore, any operation can modify these values. Therefore, the behaviour of an operation cannot be considered independent from the object for which it is defined.

Having identified the class as the basic unit of testing, we will examine a possible strategy for testing a class. Some more complete test strategies are given in (Smith and Robson, 1992) and (McGregor and Sykes, 1992).

3.1. Specification-based testing

The first step in testing a class is to write scenarios to test the possible interactions of the operations on its instances. Of course, exhaustive testing (i.e. writing all possible scenarios, the number of which is often infinite) is out of the question, therefore the tester will have to limit the number of scenarios while maximizing their efficiency.

The strategies for writing scenarios is dependent upon the availability of formal specifications of the class to be tested. If formal specifications are not available, the quality of scenarios will be determined by the common sense of the tester. C. Turner (Turner and Robson, 1992) describes a technique for writing scenarios in the context of state-based testing.

The work that has been done on the test of data types by using formal specifications (Bernot et al., 1991),(Doong, 1993) can be very useful for the specification-based testing of object-oriented software. These specifications are independent of the code of the program, which is a property fulfilling our needs since the interface of a class is independent from its implementation. Assuming that a formal specification of a class is available, these methods are very convenient, for example, to identify an ideal exhaustive test set and to (semi-)automatically generate test suites. However, it is not yet possible to give a sound basis of such testing due to the lack of a well-admitted object-oriented specification theory (Ehrich et al., 1991).

The specification-based testing should then consist in executing those scenarios and checking the behaviour of each invoked operation against its specification. (This is the role of the so-called oracle.)

3.2. Program-based testing

Procedure-based testing must be applied to the classes as such, and to their operations individually.

On the class level, measures can be computed to test to which degree the scenarios cover the reachable states of an object, or whether a sufficient number of interactions between operations has been tested. In the latter case only meaningful operations, i.e. those which have an effect on the state of the object, should be taken into account.

For the individual operations, coverage measures must be computed. Although the procedure-oriented software decomposition is incompatible with the concept of the class, some of its testing techniques are still appropriate for the testing of the operations of an object. It is possible that the implementation of a method is structurally similar to a subroutine written in a non-object-oriented language. This is often the case in languages in which the procedure-based and the object-oriented paradigm coexist, such as in Ada 9X, C++ or Modula-3. In this situation we can meaningfully apply traditional procedure-based testing methods, for example, we can compute coverage values for individual methods and for the states of a class.

However, the traditional approach should be used with care, since a program may contain different degrees of object-orientedness in different parts, and a traditional testing method that works for one part may not adequately test another part of the same program. Moreover, the traditional metrics used in procedural software may not be applicable. Complexity measures based on control flow such as McCabe's cyclomatic complexity have been known to be useful in path coverage testing (McCabe, 1976) (Fiedler, 1989). The cyclomatic complexity computes the number of independent decision paths in a program. Since exhaustive structural testing requires testing of all possible combinations of decision paths, the cyclomatic complexity measure can give an idea during a coverage test of how many tests are required for an exhaustive structural test. However, if the program is strictly object-oriented, the absence of the traditional control flow in the method implementation will prevent a meaningful application of a control flow based complexity metric to an object-oriented program.

There are currently no metrics available that adapt the classical control flow metric in a way suitable for strictly object-oriented programs. In object oriented programming, the control flow is interrupted by message passing between classes. Thus, control-flow based metrics are generally not applicable to strictly object-oriented program code. Since a class has many characteristics of a module, for example encapsulation and information hiding, the complexity metric would have to account for inter-module dependencies as a form of complexity.

There have been attempts to extend the original cyclomatic measure (Yang et al., 1991). However, the composite metrics developed to include inter-module dependencies did not satisfactorily integrate the new dependency-measure with the existing metric. This and the discrepancy between the existing control flow based complexity measures and the requirements for the adapted metric suggest that it may not be possible to successfully adapt metrics designed for monolithic programs to highly modularized, object-oriented programs. Recent research conducted with this problem in mind led to a number of metrics which do no longer measure control-flow, but the amount of dependency among program modules (e.g. objects) (Ammann and Cameron, 1994). While the control-flow based metrics assist testing by indicating the number of paths to be tested, dependency metrics may be of assistance in testing object-oriented programs by measuring the number of dependency relations among objects. The metrics implementation by Ammann and Cameron also allows for a detailed dependency analysis such that not only the number of dependencies is revealed, but also the objects which are part of a particular dependency relationship are identified. Unfortunately, problems with the analysis of dependencies can occur in the presence of polymorphism (see also section 3.3.3).

In our opinion, the development of new metrics to measure the complexity of modular, object-oriented code and which can be used for testing purposes is an important objective of future research.

3.3. Impediments to the testing of classes

Both specification- and program-based testing can be impeded by some of the mechanisms characteristic of object-oriented programming languages. The four most important examples of this effect are encapsulation, shadow invocations, dynamic binding, and conversions.

3.3.1. Encapsulation

Although encapsulation is a benefit since it increases the level of abstraction by separating the interface of a class from its implementation, it hinders testing because the resulting opacity toughens the construction of oracles: the internal properties of an object cannot be consulted, and the only way to observe the state of an object is through its operations, i.e. the testing relies on the tested software itself.

The programmer can solve this problem in two ways: the first solution is to write code that breaks the encapsulation, either by taking advantage of language features (such as the friends members of C++ or the child unit of Ada 9X) or by intrusive means (modifying the tested class or adding a subclass to introduce operations which provide the tester with visibility on the hidden features of the objects, e.g. test points (Liddiard, 1993)).

The second solution is to use equivalence scenarios (Doong, 1993): different sequences of operations which must place the object in the same state. The resulting states can be compared either by using observer operations (operations which have no impact on the state of the object), or by applying other equivalence scenarios.

3.3.2. Shadow invocations

Some operations, the construction and destruction operations, are automatically invoked at birth (declaration) and death of objects. Since these invocations are implicit, they never appear in the code and cannot be taken into account either to write scenarios, or to compute coverage values.

3.3.3. Dynamic binding and polymorphic parameters

Since dynamic binding involves that some choices regarding the execution of a program may be postponed until run-time, and cannot be statically determined, it raises a problem similar to shadow invocations with respect to the computation of coverage values: it is not possible to create test sets that will cover all the possible invocation of an operation on a polymorphic object, especially since a hierarchy of classes is freely extensible: it is possible, at any moment, to add a derived class to a hierarchy, without even requiring the recompilation of the considered operation.

A similar problem occurs when testing (specification-based and program-based) an operation one or more parameters of which are polymorphic. As testing an operation consists in checking its effects when executed for various combinations of actual parameters, a test suite must ensure that all possible cases of bindings are covered.

3.3.4. Conversions

It is possible in some languages to convert objects belonging to the root type of a hierarchy to any class in the hierarchy, in order to make the management of heterogeneous data structures easier. This conversion can provoke a failure if an object is converted to a class to which it does not belong, and is asked to select a feature or invoke an operation which is not defined for the class to which it is being converted but which exists for the class to which it belongs. Since these typing faults can usually not be caught at compile-time, care must be taken not to overlook them when designing test suites.

4. Testing hierarchies

Testing all classes individually is not sufficient as an adequate test of an object-oriented system. It is possible to program on a hierarchy-wide basis; in the alert-system example, the sensor is able to detect any alert in the *Alert* hierarchy. It is therefore crucial to test classes in the enclosing context of the hierarchy to which they belong. The main issue in testing hierarchies is to consider the compatibility level between the classes forming the hierarchy, i.e. to which extent the descendants' operations share the semantics of their parents' operations. Wegner and Zdonik have identified four different levels of incremental modifications between the classes of a hierarchy (Wegner and Zdonik, 1988):

- Inheritance with cancellation (exceptions)

  A descendant can restrict the inherited properties, thus relaxing the constraints imposed by his parent. It has no commitment to its parents' definitions, and is able to redefine or even eliminate any inherited operation.
- Name-compatible modifications (classes)

  All operations of a descendant must also be defined for its parents, but they may be overridden in the descendant. Thus, a name existing in a descendant also exists in the ancestors. Apart from the names, there is no other commitment between those operations.
- Signature-compatible modifications (signatures)

  All operations of a descendant must also be defined for its parents and each descendant operation must have a type compatible with the type specified by the parent.
- Behaviour-compatible modifications (types)

  All operations of a descendant must also be defined for its parents and the operations must have compatible behaviours. Syntax is specified by signatures and semantics by interpretations of equational axioms.

Object-oriented programming languages usually implement inheritance by choosing one of those schemes.

A way to test a hierarchy is to first test all the descendant classes individually, as if they were independent, and to examine then the adequacy of the behaviour of the descendant compared to the one of the parent depending on the allowed level of modification.

4.1. Testing subclasses individually

Since the descendant class is obtained by refinement of its parent class, it seems natural to assume that a parent class that has been tested can be reused without any further retesting of the inherited properties. However, this intuition does not hold because of possible levels incompatibilities. For example, an overridden operation may not have the same behaviour, let alone the same implementation as the one of its parent and must therefore be retested. This issue was first studied in (Perry and Kaiser, 1990) in which Weyuker's axioms for adequate testing were applied to object-oriented software and is also discussed in (Harrold et al., 1992). Hence, some of the inherited properties need retesting in the context of the descendant class. In order to retest only the properties which need retesting (incremental testing) as opposed to retesting the entire set of inherited properties we need to identify the properties which have to be retested. Identifying this minimal set of properties the behaviour of which is different from the one of the parent class may not be trivial. Here are some of the traps a tester should avoid falling into when drawing up the list of operations to be re-tested:

- Added operations

  As discussed before, the advantage of encapsulation is that clients do not have direct access to the data structure of the objects. However, inheritance breaks encapsulation: the descendant class has access to the features of the parent class and can modify them. While encapsulation builds a wall between the class and its clients, it does not prevent the descendant class to corrupt inherited features. Thus, even if the specification and the code of the inherited operation are the same in the parent and the descendant class, the increments in the descendant class can lead to changes in the execution of the inherited operations, i.e. the added operations can have an impact on the state of the object in such a way that portions of code which were previously unreachable and were therefore not tested may become reachable in the context of the subclass and thus do need retesting.

- Overridden operations

  Clearly, overriding requires retesting of the overridden operation. If a new implementation of an operation is required, the new implementation will of course not reproduce the same behaviour as the inherited code. Moreover, a side effect of overriding is that it also requires the retesting of all the operations that invoke the overridden operation as part of their implementation, no matter if they are inherited from the class in which the overridden operation was first defined or in a more direct ancestor. Since those operations make use of an operation the behaviour of which has been modified, their own behaviour is also modified and they need retesting.

- Multiple inheritance

  Multiple inheritance does not introduce problems other than the problems exposed above, unless the properties inherited from the various parent classes “step on each others feet”. This case occurs when two or more parent classes have homograph operations (i.e. with similar name and profile) and the language resolves the ambiguity of choosing by implicitly selecting which of the homograph operations is inherited by the descendant class (for example by a precedence rule like in CLOS (Kiczales et al., 1991)). The homograph operation inherited from one of the parent classes overrides the other homograph operations even in classes in which those other operations have been defined. Therefore it will be used for all invocations of this operation, thus changing the behaviour of operations which make use of it.

In the inheritance with cancellation scheme, the test suite for the descendant would have to be modified to suppress all references to the removed properties. The tester must also take care that no operation makes an invocation to a removed operation. This is easy to achieve in program-based testing, as long as no polymorphic names are put on stage.

4.2. Testing the compatibility of the behaviour

As we have seen in the alert-handling system example, it is possible to work with names (to which we will refer as polymorphic names) which can denote any object within a given hierarchy. Note that this hierarchy of classes can be as vast as the totality of all classes in the system if all classes are descendants of a unique root class (e.g. Smalltalk’s Object, Eiffel’s Any). Testing the adequacy of the behaviour of all classes in a hierarchy is important to gain confidence in an operation a statement of which is a call to a dynamically invoked operation since polymorphism brings undecidability to specification-based testing (which is the outcome of the invocation of an operation with a polymorphic name?) and program-based testing (what is the coverage of such invocations?).

Hence, it becomes necessary to make assumptions regarding the behaviour of the invoked operation. For each operation of a class in a given hierarchy, a “hierarchy specification” must be provided, which specifies the expected minimal behaviour of an operation and all its possible redefinitions. These assumptions are the assertions of the operation, i.e. the conditions that must be respected by every implementation of the operation (the original implementation and its redefinitions). Assertions can be refined for the specification of an overridden operation, but cannot be relaxed. (A mechanism to implement assertions is part of the Eiffel programming language (Meyer, 1992).)

These assertions should not be over-specified since the exact behaviour expected from an operation is usually not wanted in its redefinitions (thus the need to override the unwelcome behaviour with a proper one). Thus, the “hierarchy specification” is only a subset of the specification of the operation for the root of the hierarchy. A test suite should be designed for this hierarchy specification and each class added to the hierarchy should conform to it.

Another approach would be to use formal specifications that support the inheritance of specifications (Ierusalimschy, 1993) and use those specifications to semi-automatically generate test sets. However, this approach is only applicable for behaviour-compatible hierarchies.

5. Testing subsystems - integration testing

Integration testing is the phase in which the interactions between the various components of a system are scrutinized. This phase should also take into account the dynamic (i.e. non-structural) relationships that can exist between objects, such as associations.

Although classes are abstractions, they are not independent from each other. There is an order in which classes should be tested. Supplier classes should be tested before their clients because the client can base its behaviour on the behaviour of its supplier. For efficiency reasons, descendants should be tested after their ancestors, to take advantage of the testing already performed for the parent. However, this precedence rules should be followed with care since they are not always practical. For example, when dealing with polymorphism a descendant class could become a supplier of its parent.

Integration testing should focus on subsystems since they also have an interface which implements services that can be easily isolated and which could be tested in a manner similar to classes and objects. However, subsystems are usually only logical constructs, i.e. there is usually no syntactic construct to designate and encapsulate them. The interface of the subsystems is therefore weak. Contrary to the interface of a class, nothing prevents a user of the subsystem to bypass the interface of a subsystem and invoke a service which was not intended to be exported. The number of these ill-invocations and their combinations can of course not be tested. This is not a problem if the subsystem is only part of a system since the only invocations to be tested are the invocations actually found in the system. However, if the subsystem is intended to become a reusable component, the tester should take care to design a test suite that accounts for those misuses.

6. Conclusion

Although the test of object-oriented software has recently been recognized as an important subject of research, all dimensions of object-oriented programming relevant to testing have yet to be covered. Previous work has focused on identifying the basic unit of testing, the class, and finding the specific problems caused by encapsulation and inheritance.

We further analyzed the problems that have already been identified. We also addressed other dimensions such as subsystems and hierarchies and analyzed the problems that can arise from polymorphism with respect to testing. However, many issues in testing of object-oriented software are still open and need further research. We identified a few of special interest, such as abstract classes, subsystem testing, formal specification based testing, and complexity metrics.

The benefits from using object-oriented methods can be substantial. However, these methods do not guarantee correctness. Thus it is essential to develop a good testing methodology such that some of the current problems can be alleviated.

7. Acknowledgments

The authors would like to thank Didier Buchs for useful discussions and for constructive comments on earlier drafts of this paper, as well as the anonymous referees for their useful comments and suggestions for improvements in the paper.

The work reported in this paper is supported by the Swiss National Science Foundation (Nationalfonds), grant number 21-36038.92.

8. Bibliography

Ammann, M. M. and Cameron, R. D. (1994). Measuring program structure with inter-module metrics. To appear in *Proceedings of the 18th International Computer Software & Applications Conference,* Taipei, Taiwan. IEEE Press.

Barnes, J., Brosgol, B., Dritz, K., Pazy, O., and Wichmann, B. (1993). *Ada 9X Rationale (Draft Version 4.0)*. Intermetrics, Inc., 733 Concord Avenue, Cambridge, Massachusetts 02138, MA, USA.

Bernot, G., Gaudel, M.-C., and Marre, B. (1991). Software testing based on formal specifications: a theory and a tool. *IEE Software Engineering Journal*, 6(6):387–405.

Booch, G. (1991). *Object-Oriented Design with Applications*. Benjamin-Cummings.

Doong, R.-K. (1993). *An Approach to Testing Object-Oriented Programs*. PhD thesis, Polytechnic University.

Ehrich, H.-D., Gogolla, M., and Sernadas, A. (1991). Objects and their specification. In Bidoit, M. and Choppy, C., editors, *Recent Trends in Data Type Specification - 8th Workshop on Specification of Abstract Data Types*, volume 655 of *Lecture Notes in Computer Sciences*, pages 40–63, Douran, France. Springer Verlag.

Fiedler, S. P. (1989). Object-oriented unit testing. *Hewlett Packard Journal*, 40(1):69–74.

Harrold, M. J., McGregor, J. D., and Fitzpatrick, K. J. (1992). Incremental testing of object-oriented class structures. In *Proceeding of the 14th international conference on Software Engineering*, pages 68–79, Melbourne, Australia. ACM Press.

Ierusalimschy, R. (1993). A formal specification for a hierarchy of collections. *Software Engineering Journal*, pages 237–242.

Kiczales, G., des Riviers, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press.

Liddiard, J. (1993). Achieving testability when using Ada packaging and data hiding methods. *Ada User*, 14(1):27–32.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320.

McGregor, J. D. and Sykes, D. A. (1992). *Object-Oriented Software Development: Engineering Software for Reuse*. VNR Computer Library. Van Nostrand Reinhold.

Meyer, B. (1992). *Eiffel: The Language*. Object-Oriented Series. Prentice Hall.

Perry, D. E. and Kaiser, G. E. (1990). Adequate testing and object-oriented programming. *Journal of Object-Oriented Programming*, 2(5):13–19.

Rook, P., editor (1990). *Software Reliability Handbook*. Elsevier Applied Science.

Smith, M. D. and Robson, D. J. (1992). A framework for testing object-oriented programs. *Journal of Object-Oriented Programming*, 5(3):45–53.

Turner, C. D. and Robson, D. J. (1992). The testing of object-oriented programs. Technical Report TR-13/92, Computer Science Division, SECS, University of Durham, England.

Wegner, P. and Zdonik, S. B. (1988). Inheritance as an incremental modification mechanism or what like is and isn’t like. In Gjessing, S. and Nygaard, K., editors, *Proceedings ECOOP ’ 88*, volume 322 of *Lecture Notes in Computer Sciences*, pages 55–77, Oslo, Norway. Springer Verlag.

Yang, H., Tsujino, Y., and Nobuki, T. (1991). The complexity measure of programs based on the inter-module dependency. *Systems and Computers in Japan*, 22(13).