The Problematics of Testing Object-Oriented Software

Stéphane Barbey, Alfred Strohmeier

Swiss Federal Institute of Technology, Computer Science Department, Software Engineering Laboratory, EPFL-DI-LGL, 1015 Lausanne, Switzerland
(e-mail: stephane.barbey@di.epfl.ch)

Published In M. Ross, C.A. Brebbia, G. Staples, and J. Stapleton, editors, SQM '94 Second Conference on Software Quality Management, volume 2, pages 411–426, Edinburgh, Scotland, UK, July 26-28 1994.

ABSTRACT

Within object-oriented methods, testing has received less attention than analysis, design, and coding. However, object-orientedness does not make testing obsolete and can even introduce new problems into the production of correct programs and their testing.

Most work in testing has been done with "procedure-oriented" software in mind. Traditional methods, despite their efficiency, cannot be applied without adaptation to object-oriented systems. In an object-oriented system, the basic test unit is a class instead of a subprogram; hence, testing should focus on classes and objects. Moreover, it is not possible to test the operations of a class in isolation, as they interact with each other by modifying the state of the object which invokes them. However, some traditional techniques can still be applied, though indirectly.

Sometimes, testing object-oriented software can benefit from object-oriented technology, for instance, by capitalizing on the fact that a superclass has already been tested, and by decreasing the effort to test derived classes, which reduces the cost of testing in comparison with a flat class structure.

The object-oriented paradigm can also be a hindrance to testing, due to some aspects of its very nature: *encapsulation*, *inheritance* and *polymorphism*.

INTRODUCTION

Within object-oriented approaches, testing has received less attention than analysis, design, and coding. This lack of interest is mainly caused by a strong belief that the object-oriented technology is the ultimate answer to the software crisis, i.e. applying an object-oriented development method will eventually lead to quality code. Moreover, there is also much trust in some values of the traditional software engineering methods, i.e. applications developed with an object-oriented development method are close enough to applications developed with a traditional method that they can be tested using traditional testing methods. For example, although Rumbaugh and al.[12] assert their willingness to apply the object-oriented technology to all stages of the software development life cycle, they deny the importance of a specific testing for object-oriented software:

> *"Both testing and maintenance are simplified by an object-oriented approach, but the traditional methods used in the phases are not significantly altered.*
>
> *However, an object-oriented approach produces a clean, well-understood design that is easier to test, maintain, and extend than non-object-oriented designs because the object classes provide a natural unit of modularity."*
>
> *— J. Rumbaugh et al., [12]*

In spite of this confidence, recent studies show that "object-orientedness" is not such a silver bullet.

> *"The Self group has produced 100'000 lines of C++ and 40'000 lines of Self. We have at least as many bugs in the Self code as in the C++ code, and these bugs are usually found by testing.*
>
> *I believe that language features are no substitute for testing."*
>
> *— D. Ungar, [14]*

Object-orientedness does not by itself ensure the production of correct programs. Although an object-oriented design can lead to a better system architecture and an object-oriented programming language enforce a disciplined coding style, they are by no means shields against programmers' mistakes or a lack of understanding of the specification.

Therefore object-oriented systems still need testing. As a matter of fact, this phase of the software life cycle is even more important for object-oriented software than for traditional software, as the "object-oriented" paradigm promotes reuse: software components will be used and re-used in various contexts, even significantly different from those that the component's developer had in mind, thus the need for robust and well-tested components. In [2], Birss affirms that because of the increased level of usage, reusable components need two to four times more testing that unique components.

In this paper, we will first show why traditional structured software testing techniques are not sufficient for testing object-oriented software by comparing the fundamental differences in their architecture. We will then consider the problems introduced by some aspects of the very nature of object-orientedness:

- *Encapsulation*

  In the presence of encapsulation, the only way to observe the state of an object is through its operations; there is therefore a fundamental problem of observability.
- *Inheritance*

  Inheritance opens the issue of retesting. Should operations inherited from ancestor classes be retested in the context of the descendant class? A case analysis will show that there is no definite answer.
- *Polymorphism*

  Polymorphic names induce difficulties because they introduce undecidability in program-based testing. Moreover, erroneous casting (type conversions) are also prone to happen in polymorphic contexts and can lead non-easily detectable to errors.

THE CLASS AS BASIC TEST UNIT

The testing process

Testing is the process of finding programmers errors or program faults in the behavior or in the code of a piece of software. It is used to give confidence that the implementation of a program meets its specifications, although it cannot ensure the correctness of a program. There are several steps in the testing of a program.

The first step is usually the unit test. To perform this step the tested software is divided into the smallest possible units that can be tested in isolation, which are called basic units [3]. Two tests are usually applied to a test unit, specification-based testing and program-based testing.

- Specification-based testing, also known as black-box testing, is aimed at testing a program against its specification only, i.e. regardless of the way it is coded. It is usually accomplished by submitting a set of possible inputs, the test data, to the program being tested and by comparing the result to the specification. The decision of whether a test is satisfied or not is made by an oracle [1].

- Program-based testing, also known as white-box testing is a kind of test complementary to specification-based testing. It consists in examining the internal structure of a piece of code to select test data or to gain confidence from certain aspects of the code (e.g. coverage of statements, of paths, of conditions,...).

The other steps include integration testing, system testing, and acceptance testing. These steps are not covered by this paper.

Most work in testing has been done with "procedure-oriented" software in mind, and some good methods of testing have been developed [10]. However, those methods can not be applied as is to object-oriented software, because the architectures of those systems differ on several key issues. In this section, we will compare the architectures of procedure-oriented systems to object-oriented systems.

Procedure-oriented systems architecture

In the structured architecture, a program is decomposed functionally into subprograms, which independently implement the services of the program. The basic unit of test is a subprogram, which can consist of or make use of other (possibly embedded) subprograms. Bigger units of testing are assembled from already tested subprograms (bottom-up integration), or alternatively, subprogram stubs in a tested subprogram are replaced by subprograms to be tested (top-down integration). Data handling is accomplished by subprograms, which may not be related, and which can be scattered throughout the system, hence the difficulty in creating a complete test unit. Subprograms communicate either by passing parameters, or by using global variables.

Object-oriented systems architecture

An object-oriented system is made up of objects and classes. An object is composed of a set of features, which define its state, and a set of operations, which define its behavior. A feature can either be a value or another object. A class is a template from which objects can be instantiated. It encapsulates the definition of the properties of its instances. A class also provides hiding of some of the properties of its instances to conceal the data structure and the details of implementation. All the features of an object are usually hidden, such that the only way the state can be examined or modified is by invoking its public (non-hidden) operations. Some operations can also be hidden. They are internally used to implement the other operations. The public (non-hidden) properties of an object form its interface. The interface is a basis for a protocol which objects use to communicate with each other by requesting from an object to invoke one of its operations (or message sending). The possible result of the operation is then returned to the caller.

The class itself can also have properties. The class features are features that are shared between all instances of the class. The class operations are operations that manipulate those class features.

Classes may be related through inheritance. Inheritance is a means for incrementally building a new class (the derived class) from an existing one (the parent class) by inheriting its properties. The derived class can express differences with its parent class by modifying, and adding properties.

Classes- vs. procedure-oriented testing

The object-oriented architecture is indeed very different from the procedure-oriented architecture.

- There are no global data and data handling is not shared between units. A class contains all the properties that can affect the state of its instance.
- A class is not a testable component, i.e. it can only be tested through its instances.
- It is not possible to test the operations of a class in isolation, as they interact with each other by modifying the state of the object by which they are invoked. Control flow analysis techniques are not directly applicable, since there is no sequential order in which operations can be invoked.

  Moreover, since every object carries a state, it is impossible to reduce the testing of an object to the independent testing of operations, i.e. the tests cannot be designed, coded and performed as if the operations were "free-floating" subprograms instead of being all integrated in an object. Every operation may alter the state of the object, or even the state of the class if the latter has class variables.

We conclude that it is not possible to consider a subprogram as the basic unit of test anymore. Within object-oriented approaches, the basic unit of organization is the class construct. Testing an object-oriented system must concentrate on the state of the objects and check that the state remains consistent when executing the operations of an object. However, if not directly, traditional techniques can still be applied for some phases of the test plan, especially functional and structural testing; for instance, coverage measures can still be used to determine the effectiveness of a test suite.

Example

To be more concrete, a simple case of testing taken from [10] will be examined.

> *"The program reads three integer values from a card. The three values are interpreted as representing the lengths of the sides of a triangle.*
>
> *The program prints a message that states whether the triangle is scalene, isosceles, equilateral."*
>
> *— G. J. Myers, [10]*

What Myers had in mind when suggesting this exercise to his readers was to show the difficulty of writing a test case that would adequately test a unit such as the following procedure-oriented unit.

```
type
    Kind: (scalene, isosceles, equilateral, error);

function  Kind_of (Length1, Length_2, Length_3: Integer): Kind
begin
   ...
end ;
```

Figure 1: "procedure-oriented" test unit

In the context of object-oriented programming, the unit to test would probably more look like the following class (in a pseudo object-oriented language).

```
class interface  Triangle
   inherit  Shape  is

   operation   Create (Length1, Length_2, Length_3: Integer)
       return  Triangle;
   operation   Length_of (Side: Integer    range  1..3)
      return  Integer;

   enumeration    Kind  is  (scalene, isosceles, equilateral, error);
   operation   Kind_of  return  Kind;

   operation   Homothety (Factor: Positive);
   operation   Similar (A_T riangle: T riangle)  return  Boolean;
end  Triangle.
```

Figure 2: "object-oriented" test unit

The noticeable changes from the first architecture to the second are:

- encapsulation/hiding

  All the operations and related types, even some which are not useful (like *Homothety* or *Similar*) to solve the problem, are encapsulated in the class. The operation *Kind_of* is not an isolated algorithm. It cannot be tested without being invoked by an instance of *Triangle*, the data structure of which is hidden. The only way to access it is by invoking selector operations defined in the interface, such as *Length_of*.

- inheritance

  The class *Triangle* inherits from the class *Shape*. *Triangle* can therefore rely on *Shape* for both its representation (data structure) and its behavior (operations implementation).

- operations binding

  The operations of the class *Triangle* are tightly bound: it is not possible to apply the operations *Length_of* and *Kind_of* to an object triangle before *Create* is invoked to create an instance. Hence, it becomes necessary to monitor the changes that take place in the features according to the operations. Therefore, testing a class requires writing scenarios to test the possible interactions of the operations onto its instances. Of course, exhaustive testing (i.e. writing all possible scenarios, which is often infinite) is out of the question, therefore the tester will have to limit the number of scenarios while maximizing their efficiency. C. Turner describes a technique for writing scenarios in [13].

Besides these technical issues, another reason to consider the testing of object-oriented software different from the traditional testing is the move in the software life cycle. The traditional development life cycle (waterfall) is challenged by continuous development methods (prototyping). Testing does not occur anymore as the steady last step in the development, but as a step that occurs repetitively for every refinement of a software component. To be efficient, a tester has to write flexible test code and stress incremental testing, to make a maximum reuse of existing test code.

ENCAPSULATION

The notion of class is a blessing for testing, since this notion involves encapsulation. The test unit is clearly designed, which effectively simplifies the testing because the determination of a test unit becomes easier. The class can thus be tested in isolation of the rest of the system. i.e.

the context in which the test is made does not affect the testing of the class.

Hiding

Hiding is a fundamental property of object-oriented programming. Programmers do not need to worry about the internals of a class, since they only use the interface to communicate with the objects. However, the resulting opacity hinders testing, since the coherence of the state of the object cannot be checked after invoking an operation of the class. In the presence of *hiding*, the only way to observe the state of an object is through its operations; there is therefore a fundamental problem of observability, as part of the testing relies on the tested software itself.

There are many workarounds to that problem.

The first and most obvious solution is to modify the tested class to add operations which provide the tester with visibility on the hidden features of the objects, e.g. test points [6]. This solution is usually not satisfactory since it is intrusive and it involves an overhead on the tested class. Moreover, it is not guaranteed that the class will have the same behavior in the presence of the tested code.

A refined approach is to define the added operations in a descendant class, which is derived from the tested class, and serve to test the derived class. This method captures the functionality of test points while being non-intrusive. However, this method is useless if the derived class does not have complete visibility on the state of the inherited properties. For example, the internal (called private) properties of a C++ class are not visible to its descendants.

Besides those two general methods, several programming languages support language-specific mechanisms to break encapsulation by providing either:

- family-related constructs

  C++ has intrusive friends subprograms to define operations that are not operations of the class but have visibility on all its features. The child units of Ada 9X are non-intrusive package extensions, which have a complete view onto the private part of their parent package.
- low-level constructs

  Smalltalk's inspectors and Eiffel's class Internal provide low-level operations to examine all the features of an object. These operations break encapsulation by accessing the physical object structure.

- unchecked type conversion

  If the type system of the programming language is weak, or if the language provides unchecked type conversion, it is possible to break encapsulation by writing another class, the data structure of which is a clone of the tested class except that all its properties are public and by casting instances of the tested class to instances of the clone class to access their features freely .

Of course, if there is no way to break the encapsulation, or if the data structure is inaccessible (e.g. because it is not part of the available specification), the tester will have to gain confidence in the selector operation before testing the other operations. The tester will have to take this order of precedence into account when elaborating scenarios.

Non-instantiatable classes

On top of the problem of accessing the state of an object is the problem of testing non-instantiable classes. Non-instantiatable classes are classes from which instances cannot be created because they do not contain enough information.

There are three kinds of non-instantiatable classes: abstract classes, the implementation of which is not completely defined (and must be subsequently defined in derived concrete classes), generic (parameterized) classes (some components of which are unspecified to be more general, for instance the type of the items to be put in a class Stack), and mixin classes (abstract subclasses that can be derived from any base class to create a new class).

Since instances of those classes cannot be created, it is impossible to test them as is. The tester will therefore have to develop a minimal test suite that provides the different bindings for the lacking information to achieve an exhaustive test of those non-instantiatable classes. This topic is still a subject of research in which further work is needed.

INHERIT ANCE

Inheritance is a mechanism that allows for a class, the derived class, to inherit properties, features and operations, from one (single inheritance) or many classes (multiple inheritance), its base classes. The derived class can then be refined by modifying or removing the inherited operations, or adding new properties.

Since the derived class is obtained by refinement of its parent class, it seems natural to assume that a base class that has been tested can be reused without any further retesting of the inherited properties. This intuition is however proved false in [1   1], with the help of W   eyucker 's axioms for adequate testing: some of the inherited properties need retesting in the context of the derived class. Therefore, to take advantage of the testing already completed for the base class to test the derived class, i.e. to do incremental testing, and not to retest the whole set of inherited properties, we must diagnose which of those properties need retesting, i.e. the minimal set of properties the behavior of which is different from the parent class.

As the inheritance scheme differs from one language to another , there is no language-independent method to define this set. In the following discussion, we will examine the most commonly found schemes. An algorithm that determines this minimal subset for the C++ programming language can be found in [4].

Strict inheritance

Strict inheritance is the simplest scheme of inheritance. A derived class is a strict heir of its parent class if it keeps the exact inherited behavior of its parent. The inherited properties cannot be modified (e.g. overridden); the derived class can only be refined by adding new properties.

Once more, despite intuition, strict inheritance involves retesting of some of the inherited properties. As discussed before, the advantage of encapsulation is that clients do not have direct access to the data structure of the objects. Inheritance does however break encapsulation: the derived class has access to the features of the base class, and can modify them. While encapsulation builds a wall between the class and its clients, it does not prevent the derived class to mess up with the inherited features.

Thus, although the specification and the code of the inherited operation are the same in the base and the derived class, the increments in the derived class can lead to changes in the execution of the inherited operations: the added operations can have an impact on the state of the object, so that portions of code that were previously unreachable, and were thus not tested, may become reachable in the context of the subclass, and thus need testing.

Subtyping

The second, and most common, scheme of inheritance is subtyping. Besides the properties of strict refinement, subtyping allows for the redefinition (overriding) of the inherited properties (i.e. to give a new implementation to an inherited operation, to be used in the derived class). Overriding occurs when the behavior of an inherited operation is not appropriate in the context of the derived class.

Overriding does of course imply retesting the overridden operation. If the programmer has felt the need to give a new implementation to an operation, the latter will not reproduce the exact behavior of the inherited code. Moreover, a side effect of overriding is that it also implies 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 latter subclass in the inheritance hierarchy: since those operations make use of an operation the behavior of which has been modified, their own behavior is also modified and they need retesting.

```
with Coordonates; use Coordonates;
with Displays; use Displays;
package Buttons is

   type Button is tagged private ;
   procedure Redraw (B: Button; C:Coord);
   procedure Choose (B:Button);

   type PushButton is new Button with private ;
   procedure Redraw (P: PushButton; C:Coord);

end Buttons;
```

Figure 3: Operation Overriding in Ada 9X

In the above example (figure 3), inspired from Interviews [7], a class Button has two operations, Redraw and Choose, the implementation of which makes use of Redraw. In the class PushButton, which is derived from Button, the operation Redraw is overridden, but not Choose. However, since the implementation of Choose makes use of an overridden operation, it will have to be retested, although its code is not modified.

Subclassing

Subclassing is a scheme of inheritance in which the derived class is not considered as a specialization of the base class, but as a completely new abstraction which bases part of its behavior on a part of another class. This scheme is also called implementation inheritance.

The derived class can therefore choose not to inherit all the properties of its parent. As a consequence of subclassing, the test suite defined for the subclass must therefore be modified to suppress all references to the removed properties. The tester must also take care that no operation makes a call to a removed operation. This is easy to achieve in program-based testing.

Subclassing can also introduce other problems that will be discussed in the section on polymorphism.

Multiple inheritance

Multiple inheritance does not introduce problems other than the problem exposed above, unless the properties inherited from the various base classes "step on each others feet".

This case occurs when two or more base classes have homograph operations (i.e. with similar name and profile), and the language resolves the ambiguity of choosing by implicitly selecting which one of the homograph operation is inherited for the derived class (for example by a precedence rule like in CLOS [5]).

```
; definition of class first
(defclass first-class
    ()
    (first-slot)   )

(defmethod a-method ((an-object first-class))
    (format t "Method of First"))

(defmethod first-method ((an-object first-class))
    (a-method an-object))

; definition of class second
(defclass second-class
    ()
    (second-slot))

(defmethod a-method ((an-object second-class))
    (format t "Method of Second"))

; definition of class third
; (multiple inheritance of second-class and first-class, nothing else)
(defclass third-class
    (second-class first-class)
    ())
```

Figure 4: A case of multiple inheritance in CLOS

The homograph operation inherited from one of the base classes overrides the other homograph operations, even in the classes where those other operations have been defined, Therefore, it will be used for all calls of this operation, changing the behavior of operations that makes use of it.

In the above example (figure 4), a class third-class is built by deriving from first-class and second-class. Both first-class and second-class have an homograph operation a-method. In the class third-class, the operation first-method, which is inherited from first-class, and used to call a-method defined in first-class will call a-method defined in second-class. Therefore, any call to first-method with an instance of third-class will result in outputting “Second Method”.

Note that in most object-oriented languages that support multiple inheritance (C++, Eiffel), such ambiguities must be explicitly resolved by the programmer himself.

POLYMORPHISM

Polymorphism is the possibility for a name, i.e. a variable or a reference (called polymorphic name below) to denote instances of various classes. It is usually constrained by inheritance, in which these various classes must belong to a hierarchy of classes made up of a root class or its descendants. If the inheritance scheme is subtyping, the denoted objects all have at least the properties of the root class of the hierarchy. Thus, an object belonging to a derived class could be substituted into any context in which an instance of the base class appears, without causing a type error in any subsequent execution of the code. (This principle is known as Liskov’s substitution principle [8]). 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).

We will now examine different uses of polymorphism that can affect the correctness of a program and cause trouble to testing.

Undecidability of dynamic binding

Polymorphism brings undecidability to program-based testing. Since polymorphic names can denote objects of different classes, it is impossible, when invoking an operation of a polymorphic name, to predict until run-time, what the code is, that is about to be executed, i.e. whether the original or a redefined implementation will be selected. (This is called dynamic binding.)

Therefore, to gain confidence in an operation a statement of which is a call to a dynamically invoked operation, it becomes necessary to make assumptions regarding the nature of the polymorphic parameters, i.e. a "hierarchy specification" must be provided for each of their operations, which specifies the expected minimal behavior of their operation and all their possible redefinitions. This "hierarchy specification" is only a subset of the specification of the operation: the exact behavior expected from an operation is usually not wanted in its redefinitions (thus the need to override the unwelcome behavior with a proper one).

These assumptions are the assertions of the operation: 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 [9].)

Extensibility of hierarchies

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.

However, given a polymorphic call or an operation with one or more polymorphic parameters, it is impossible to plan a test in which you check the operation with parameters of all the possible classes, because a hierarchy of classes is freely extensible: it is possible, at any moment, to add a derived class to a hierarchy, without even causing the recompilation of the considered operation.

In this case too, the use of assertions can facilitate the task of the testers.

Heterogeneous containers and type casting

Heterogeneous containers are data structures the components of which may belong to various classes, constrained in the same way as polymorphic names.

However, some objects do not have the same set of properties as the root class of the hierarchy to which they belong. To let those objects make full use of their properties, it is possible to cast the objects contained in the heterogeneous data structures to any class in the hierarchy.

This can lead to two common faults:

- an object being casted to a class to which it does not belong, and being therefore unable to select a feature or invoke an operation, because it lacks these properties. This is called downcasting.
- if the scheme of inheritance is subclassing, an object being not casted to its class and being compelled to invoke a message that was removed for its class.

Since these typing faults can usually not be caught at compile-time, care must be taken when designing test suites not to overlook them.

CONCLUSION

This paper shows that, although the test of object-oriented software can resort to some techniques developed for the test of traditional procedure-oriented software, the differences that arise between those two paradigms are important enough to develop a test methodology specific to object-orientation. It also show that its nature: encapsulation inheritance, polymorphism conveys identifiable, albeit nontrivial, problems.

We should not dismiss the importance of testing. The more object-oriented technology gains recognition, the more components repository will grow , gathering components developed in various environments. If we want reuse to finally occur , programmers must be daring enough to import those components in their applications, without being afraid of a possible lack of correctness, thus the need to induce confidence in those components by testing them meticulously .

ACKNOWLEDGMENTS

This work is supported by the Swiss National Science Foundation (Nationalfonds), grant number 21-36038.92.

REFERENCES

[1] Gilles Bernot, Marie-Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *IEE Software Engineering Journal*, 6(6):387–405, November 1991.

[2] Bob Birss. Testing object-oriented software. *SunProgrammer - The Newsletter for Professional Software Engineers*, 1(3):15–16, Fall/Winter 1992.

[3] Steven P. Fiedler. Object-oriented unit testing. *Hewlett Packard*

Journal, 40(1):69–74, April 1989.

[4] Mary Jean Harrold, John D. McGregor, and Kevin J. Fitzpatrick. Incremental testing of object-oriented class structures. In *Proceeding of the 14th international conference on Software Engineering*, pages 68–79, Melbourne, Australia, May 11-15 1992. ACM Press.

[5] Gregor Kiczales, Jim des Riviers, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991. 335 pages.

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

[7] Mark A. Linton, Paul R. Calder, and John M. Vlissides. Interviews: A C++ graphical interface toolkit. Technical Report CSL-TR-88-358, Stanford University, July 1988.

[8] Barbara Liskov, Russel Atkinson, Toby Bloom, Eliot Mogs, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Sciences*. Springer Verlag, Berlin Heidelberg New York, 1981.

[9] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, 1992.

[10] Glenford J. Myers. *The Art of Software Testing*. Business Data Processing: a Wiley Series. John Wiley & Sons, 1979.

[11] Dewayne E. Perry and Gail E. Kaiser. Adequate testing and object-oriented programming. *Journal of Object-Oriented Programming*, 2(5):13–19, January 1990.

[12] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1991.

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

[14] David Ungar. Position paper for the OOPSLA ’93 workshop on object-oriented testing. September 1993.

RESER VOIR

Even worse, the object-oriented paradigm can also be a hindrance to testing, due to

Testing object-o riented software is now emerging as a new field of research. Those efforts have not yet solved all problems and many issues are still open. Some of them will be studied in this paper .

terminology client/supplier , ancestor (parent), descendant, selector operation.

Polymorphic calls (dynamic binding) are difficult to test.

, which we wil call a class hierarchy specification

Moreover , it becomes very difficult to check the behaviour of such an operation. W e have already avouched that the test suite needed for an overridden operation is different from the test suite used for the original operation defined in the parent. In the case of a dispatching call, this brings the difficulty to know which is the expected behaviour , and which suite must be chosen to test whether the operation is executed without fault or not.

Moreover , when extending a software by a new derived class (programming by extension), calls with possible bindings to operations of this new derived class must be re-tested.

Extensibility

The problem gets even worse when dealing with multimethods (dispatching operations with two controlling operands of different types): the number of possible dispatching cases grow (number of possible cases for the first operand multiplied by the number of possible cases for the second operand).

An abstract class provides the interface of a class the data representation of which is incomplete, and for which some operations, the abstract operations, have no given implementation. Such an interface can be shared by many different implementations of the same abstraction by providing complete implementations in derived classes. For example, an abstract class

The given implementations are usually built on the abstract operations.

To be instantiatable, the implementation of an abstract class must be completed by its descendants.

- parameterized classes (generic classes)

  A parameterized class is a class some properties of which must be given by its user. These properties are usually left to the user to make the class more general and reusable. For example, a class Stack is usually implanted as a parameterized class, for which the programmer must provide the type of the elements it will contain.

- mixin classes

  A mixin class is a special case of parameterized class, where the generic parameter is the parent class.The parent class can be either or constrained to a specific hierarchy of types, in which case it is called a constrained mixin. The purpose of mixins is to be used to specialize the behavior of any class. For example, a mixin *Version* could be used to add version management to different classes, such as *Document* or *File*.