Open Issues and Research Directions in Object-Oriented Testing

Alessandro Orso *Sergio Silva*[*]

Politecnico di Milano – Dipartimento di Elettronica e Informazione, Milan, Italy

Abstract

To date, research in the field of object-orientation is mostly focused on the first phases of the traditional software life-cycle. A big effort has been spent on methodologies and techniques for specification, design and implementation of object-oriented systems. On the contrary, the research in the testing of this kind of system is still in an early phase.

Often, the approach followed when testing object-oriented systems is just to apply well-known traditional testing techniques, neglecting the big differences related with this new paradigm of programming. Aspects like information hiding, encapsulation, polymorphism, dynamic binding, inheritance and genericity, typical when not peculiar of object-oriented languages, lead to the need of defining new approaches as far as testing is concerned.

The main goal of this article is to classify the problems related with the testing of object-oriented system and to illustrate the approaches proposed so far in literature for solving them.

1 Introduction

Object-oriented development methods are becoming more and more popular and object-orientation is viewed by many people as the silver bullet for solving software related problems like maintenance, reuse, and quality. In the last decade the research in this field has grown considerably. A lot of effort has been put in the definition of methodologies and techniques for the specification and design of object-oriented systems [3, 2, 28, 5].

The underlying idea seems to be such that the use of the object-oriented approach together with an ad hoc specification and design methodology can automatically lead to good and error-free software. Unfortunately, while these approaches effectively lead to a better quality software, enforcing modular architectures, ease of reuse and good documentation, they do not provide correctness by themselves. The production of software is carried out by humans and human beings are and always will be error-prone. Error free software is still a chimera, and there will always be the need for testing.

Furthermore, object-oriented software is harder to test than traditional procedural software. Traditional testing techniques can not be applied tout-court to object-oriented

[*]Partially supported by CONACyT-Mexico

software. We need to adapt these techniques, or even to define new techniques, which are able to cope with the peculiarities of object-orientation.

In this paper, we examine problems introduced by object-orientation in the field of testing. We illustrate how information hiding, encapsulation, polymorphism, dynamic binding, and inheritance may lead to new testing problems unseen with traditional software and we present the different approaches found in literature to solve such problems. Section 2 illustrates the aspects of object-oriented languages we consider in this paper; Section 3 examines the impact of object-oriented features on testing; Sections 4 to 9 analyze the different approaches present in literature to cope with the problems introduced by object-oriented technology to the testing activities; finally, Section 10 draws some conclusions.

2 Object-orientation aspects

In literature several different definitions of object-orientation can be found and the term *object-oriented* is often used to describe a lot of different approaches and/or techniques. Object-oriented technology comprises specific analysis, design and implementation methods. Since this is a paper on testing, and testing is strongly linked to the implementation, the analysis performed in this paper often refers to object-oriented specific features of programming languages. In the following we introduce the characteristics we consider as essential for an object-oriented language:

Data Abstraction: Objects are described as implementations of abstract data types (ADTs). An object-oriented language must support abstract data type definitions. Usually, an ADT definition is called *class*, while an *object* is a run-time instance of a class

Inheritance: Inheritance is a key concept for object-orientation. Object-oriented languages must allow for defining an abstract data type deriving it from an existing one (either extending it or restricting it)

Polymorphism: Program entities should be permitted to refer to objects of more than one class, when a hierarchical relationship among these classes exists.

Dynamic binding: Operations applied to a polymorphic variable should be permitted to have different realizations and the identity of such operations must be resolved dynamically based on the type of the object the variable is referring to.

In the rest of the article we consider these aspects separately to identify the different problems every one of them can introduce as far as testing is concerned. In addition, there are two other aspects that are not purely related to object-orientation, but which are present in the most common object-oriented languages and which have a heavy impact on testing:

Genericity: Abstract data types can be defined in a parametric way, i.e., as template classes whose data structure and operations are defined in terms of classes or operations specified in the formal parameters list. Generic classes are not used to generate objects, but to generate new classes

Exception handling: To improve reliability, it is necessary that erroneous conditions can be recognized by the program and that certain actions are executed in response to the error.

For the sake of brevity, these two topics are not addressed in this paper. The interested reader may refer to the extended version of this article [25].

3 Problems with object-oriented testing

From a technical perspective, object-oriented systems are similar in many respects to traditional systems. Both at the system level and at the procedure level, the testing of object-oriented systems is much the same as the testing of traditional systems. In addition, in object-oriented programming procedures tend to be small, thus testing at this level tends to be less expensive for object-oriented systems than for traditional procedural ones. However, the complexity has moved from within code modules to the interfaces between them, making integration testing necessarily more expensive.

It is in the middle levels of testing that object-oriented systems are most different from traditionally developed systems. These differences are related with the peculiarities of object-oriented languages illustrated in the previous section. Such topics heavily impact on testing raising a set of problems only partially present, when not totally absent, in the testing of systems developed with traditional procedural languages. In this paper we classify the different problems directly related to object-oriented testing (OOT) into a set of categories. Every category is named after the object-oriented issue originating the problem when possible, otherwise a specific name is used. We identify six different areas of possible problems: testing levels, information hiding, shadow invocations, polymorphism and dynamic binding, conversions, and inheritance.

In the following section the aforementioned problems are specifically addressed and the solutions for solving them proposed so far in literature are presented.

4 Testing levels

Before starting illustrating the problematics of object-oriented testing we want to spend some words on the definition of the different testing levels in the object-orientation field. It is not straightforward to make object-oriented systems fit the traditional testing levels [13]. In fact, in literature there is not a general consensus on what should be considered a unit and what should be considered a module in an object-oriented program.

One approach considers the single operation of a class as the elemental unit of testing, and the whole class as the traditional module. Another approach considers the testing of a single operation as meaningless, since it does not take into account the state of the class. For the sake of clarity, in this paper we use the following terminology:

basic unit testing: testing of a single operation of a class (*intra-method testing*)

unit testing: testing of a class as a whole (*intra-class testing*)

integration testing: testing of the interaction among classes (*inter-class testing*)

regression testing: testing of the system as a whole when some change has been performed on one or more classes

A different classification can be found in [29] which also introduces *cluster level* testing, where a cluster is considered to be a set of classes providing a particular set of functionalities.

Please notice that we have defined only the levels of testing affected by the use of an object-oriented language. We have omitted, for example, system testing, since the testing of an object-oriented system as a whole can be performed with traditional techniques.

Many papers analyzed during this work are mostly (when not only) addressing the unit testing level [18, 11, 4, 33, 9, 12].

5 Information hiding

In traditional procedural programming the basic component is the subroutine and the testing method for such component is input/output based[14, 34]. In object-oriented programming things change. The basic component is represented by a class, where a class is composed by a *data structure* and a *set of operations*.

Objects are run-time instances of classes. The data structure defines the state of the object which is modified by the class operations (*methods*). In this case, correctness of an operation is based not only on an input/output relation, but also on the resulting state. Moreover, the data structure is in general not directly accessible, but can only be accessed using the class *public* operations.

Encapsulation and information hiding make it impossible for the tester to check what happens inside an object during testing. Due to data abstraction there is no visibility of the insight of objects. Thus it is impossible to examine their state. Encapsulation implies the converse of visibility, which in the worst case means that objects can be more difficult, or even impossible to test.

Encapsulation and information hiding raise the following main problems:

1. Problems in identifying which is the basic component to test.
2. Problems introduced by opacity in the construction of oracles: ingeneral it is not enough to observe input/output relations, but it is necessary to take into account the state of objects; on the other hand, the state of an object can be inaccessible, being observable only through class methods (thus relying on the tested software itself).

Moreover, the possibility of defining non-instantiable classes (e.g., abstract classes, generic classes, interfaces) introduces additional problems related to their non straightforward testability.

Figure 1 illustrates an example of information hiding. This simple example shows how the testing of method `checkPressure` in isolation is meaningless.

There are fundamentally two approaches proposed in literature for testing object-oriented programs as soon as encapsulation and information hiding are concerned:

```
class Watcher {
  private:
    int status;
  public:
    void checkPressure() {
      if(status==1) {...}
      else if(status==2) {...}
      else ...
  }
...
};
```

Figure 1: An example of information hiding

Breaking encapsulation: it can be achieved either exploiting features of the language (e.g., the C++ *friend* [30] construct or the Ada *child unit* [19]) or instrumenting the code. This approach allows for inspection of private parts of a class. The drawback in this case is the intrusive character of the approach. An example of this approach can be found in [22].

Equivalence scenarios: this technique is based on the definition of couples of sequences of method invocations. Such couples are augmented with a tag specifying whether the two sequences are supposed to leave the object in the same state or not. In this way it is possible to verify the consistence of the object's state by comparison of the resulting states instead of directly inspecting the object's private parts. Moreover, in the presence of algebraic specifications this kind of testing can be automated. The advantage of this approach is that it is less intrusive than the one based on the breaking of encapsulation. However, it is still intrusive, since the analyst needs to augment the class under test with a method for comparing the state of its instances. The main drawback of this technique is that it allows for functional testing only. Moreover, the fault hypothesis is non-specific: different kind of faults may lead to this kind of failure and many possible faults may not be caught by this kind of testing.
Equivalence scenarios has been introduced in [9]. Another application of this approach is found in [32]. In this case, when testing a class, states are identified by partitioning data member domains. Then, interactions between methods and state of the object are investigated. The goal is to identify faults resulting in either the transition to an undefined state, or the reaching of a wrong state, or the incorrectly remaining in a state.

6 Shadow invocations

Shadow invocations are operations automatically or implicitly invoked. Such invocations do not appear in the code (i.e., there is no explicit invocation of the operation in the program). Examples of operations often invoked in this way are constructors, destructors, and conversion operators.

Shadow invocations introduce problems when constructing scenarios and when computing coverage values.

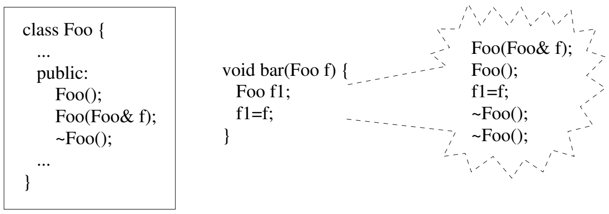


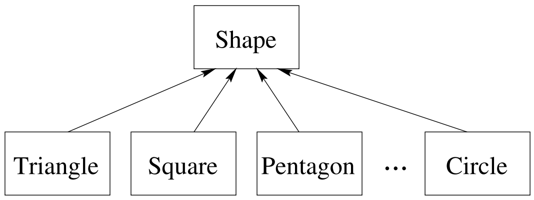
Figure 2: An example of shadow invocations

Figure 2 shows an example of implicit invocations of methods when method `bar` is invoked. In this case the two lines of code composing the procedure spread as shown on the right part of the figure: two `Foo`'s constructors are invoked and `Foo`'s destructor is invoked two times.

So far the only reference to the shadow invocation problem we found is the one in [1].

7 Polymorphism and dynamic binding

The term polymorphism refers to the capability for a program entity to dynamically change its type at run-time. This introduces the possibility of defining polymorphic references (i.e., references that can be bound to objects of different types). In the general case the type of the referred object must belong to a type hierarchy. For example, in C++ or Java [15] a reference to an object of type `A` can be assigned an object of any type `B` as long as `B` is either a heir of `A` or `A` itself.




```
Void foo(Shape polygon) {
    ...
    area=polygon.area();
    ...
}
```

Figure 3: A simple example of polymorphism

In the presence of polymorphism a reference is assigned both a static and a dynamic type. The static type of a reference is the type it is given in its declaration, while the dynamic type of the reference depends on the type of the object the reference is referring to at run-time.

A feature closely related to polymorphism is *dynamic binding*. In traditional procedural programming languages, procedure calls are bound statically. Roughly speaking, the actual code associated to a call is known (i.e., uniquely identified) at link time. In the presence of polymorphism, the precise object that actually processes a message is not pre-determined, but it must be an instance of one of a finite number of classes. More precisely, the actual code invoked as a consequence of a message invocation on a reference depends on the dynamic type of the reference itself . For example, if class `derived` is a sub-class of class `parent`, the method `X` (defined in class `parent`) is re-defined in class `derived`, and `obj` is a reference to an object of type `parent`, then when `X` is invoked on `obj` it is impossible to statically identify which definition of `X` will be executed. In addition, a message sent to a reference can be parametric, and parameters can be at their turn polymorphic references.

Late binding can easily lead to messages being sent to the wrong object. It can be very difficult to first identify and then test all the possible bindings. A polymorphic re-declaration can radically change the semantics of a method and thus fool the clients of the class it belongs to. Since sub-classing is not inherently sub-typing, dynamic binding on an erroneous hierarchical chain, can produce undesirable results. Moreover, even when the hierarchy is well formed, errors are still possible, since the correctness of a redefined method is not guaranteed by the correctness of the superclass method.

When performing multi-class testing and coverage is concerned, the problem of defining the sufficient coverage for a polymorphic call arises. Often it is infeasible to perform an exhaustive testing stressing all the possible combinations "sender of the message"- "receiver of the message". It is necessary to define a criterium allowing the tester to chose only a sufficient/optimal subset of all the possible combinations. The trade-off here is between the possible infeasibility of the approach and its incompleteness.

Moreover, further concerns are introduced in case classes to be tested belong to a library instead of belonging to a specific system. Classes built to be used in one specific system can be tested by restricting the set of possible combinations to the ones identifiable analyzing the code [26]. Re-usable classes need a higher degree of polymorphic coverage, because such classes will be used in different and sometimes impredictable contexts. The problems introduced by polymorphism can be summarized as follows:

- Due to the combinatorial number of cases to test when testing in the presence of polymorphism, program based testing may become infeasible.
- A new definition of coverage must be provided to cope with the testing of operations on a polymorphic object.
- The creation of test sets to cover all possible calls to a polymorphic operation can not be achieved with a traditional approach.
- The presence of polymorphic parameters introduces additional problems for the creation of test cases.

Figure 3 shows a simple Java example of method invocation on a polymorphic object. In the proposed example it is impossible to say at compile-time which implementation of the method `area` is actually called.

Figure 4 illustrate a method invocation (represented in a message sending fashion), where both the sender and the receiver of the message are polymorphic entities. In addition, the message has two parameters, also polymorphic. The one represented in this latter example is the worst situation for testing purposes. As represented in the figure, the number of possible situations (type of the sender, type of the receiver and type of parameters) is combinatorial. Moreover in this example we do not take into account the states of the objects involved. The different objects may behave differently depending on their state, and this leads to a further explosion of the number of test cases to be generated.

The exhaustive testing of all the polymorphic run-time bindings is not possible in general. So far, the underlying idea when testing in the presence of polymorphism is to identify a criterium for selecting a subset of all the possible combinations of sender, receiver and parameters.

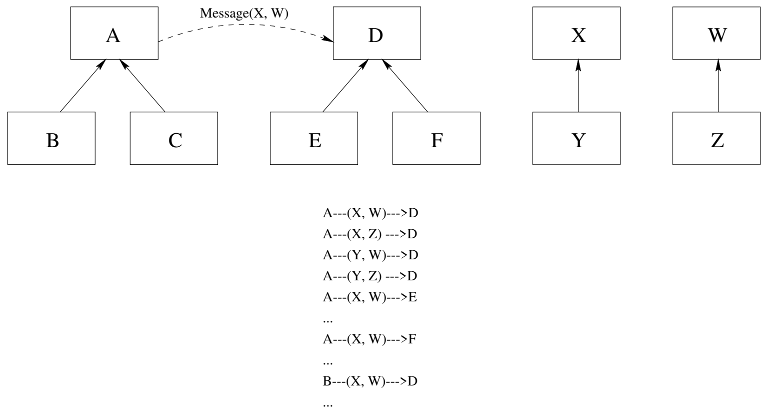


Figure 4: An example of polymorphic invocation

Problems introduced by polymorphism as far as testing is concerned, have not been addressed deeply by the different authors working on object-oriented testing. In the following we describe works developed on this topic: [17] briefly introduces the problem, but does not provide any technique for addressing it.

The work reported in [21] proposes a partial solution based on sequences of messages. Two sequences are defined: Method Sequence Specification (MtSS), that specifies the order in which the methods of a class can be invoked and Message Sequence Specification (MgSS), which specifies the causal order in which messages can be sent to instances of different classes. Both sequences are used for test case generation.

[23] provides a deeper analysis of the problem and illustrates a solution based on orthogonal arrays. The approach concentrates on message exchanges between objects, considering all the possible combinations of type and status of the sender, of the receiver and of possible parameters. Such combinations are then reduced using a mathematical method based on orthogonal arrays. This method ensures that all the pairwise combinations are excercised. The main drawback of the approach is that, because of its static nature, the method considers all possible combinations (including infeasible ones).

[27] approaches the problem from a different point of view, addressing polymorphism related issues from the integration testing perspective. This paper introduces a representation of object-oriented systems defined enriching Objectcharts [7] and then presents an algorithm that allows the analyst to identify the order in which to perform class integration testing on a pairwise basis. The article then introduces some heuristics to reduce the number of test runs.

8 Conversions

Some object-oriented languages allow the programmer to *cast* objects. In these languages it is possible to perform type conversions which can not be checked at compile-time. Casting errors may cause failures, unless the run-time system provides a way to catch and handle them. Unfortunately, there are languages (e.g., C++) where this kind of errors are hardly detected at run-time, thus they often result in program terminaton with a run-time error.

Test suites should be designed, which take into account possible casting errors.

```
Stack myStack;
...
Shape shape=myStack.top();
((Circle)shape).radius=28;
...
```

Figure 5: An example of a risky conversion

Figure 5 shows an example of a possibly wrong casting. If the dynamic type of object `shape` is different from *Circle* a failure occurs at run-time. So far, no authors addressed the problem of wrong conversions as far as the dynamic testing of object-oriented programs is concerned. Some work has been done both on static analysis of object-oriented programs in order to achieve static type correctness [6, 20] and on enrichment of the programming language to make it strongly-typed [8, 31, 24]

9 Inheritance

In traditional procedural programming languages code is structured in subroutines, which are possibly enclosed in modules. Modules are composed in bottom-up or top-down hierarchies. In this situation, when a subroutine has been tested and accepted there is no need to re-test it.

Object-oriented programs are structured in classes and inheritance is the fundamental relationship among classes. Inheritance is probably the most powerful feature provided by object-oriented languages. By taking advantage of the inheritance mechanism, programmers may create new classes that extend existing ones. As a result, classes defined using object-oriented languages are organized in a hierarchy (usually called the *is-a* hierarchy) originated by the inheritance relationship. Object-oriented languages may provide single or multiple inheritance. Object-oriented languages providing *single inheritance* allows a class to inherit from a single *parent* class. Conversely, in Object-oriented languages providing *multiple inheritance* each class may inherit from more than one parent.

The inheriting *derived* classes are allowed to perform one or more of the following operations: *override* (i.e., redefine) the inherited methods by providing a different implementation which better fits their needs, *add* new features to the inheriting class, and *eliminate* some of the features provided by their super class(es). In this paper we refer to the following inheritance schemes:

Strict inheritance : in this schema, methods can be added in subclasses

Subtyping inheritance : this schema extends the *strict inheritance* giving subclasses the possibility of overriding (i.e., redefining) methods inherited from their superclass(es)

Subclassing inheritance : In addition to the features provided by *Subtyping inheritance*, in this schema it is possible to eliminate methods from the superclass(es)

In most object-oriented languages, inheritance is a mechanism allowing for achieving two goals: code reuse and definition of type/subtype relationships. In the first case, the use of inheritance raises the following issues:

Initialization problems: it is necessary to test whether a subclass specific constructor (i.e., the method in charge for initializing the class) is correctly invoking the constructor of the parent class.

Semantic mismatch: in case of subtyping inheritance, methods in the subclasses may have a different semantics and thus they may need different test suites [29]. Moreover, multiple inheritance and repeated inheritance can lead to particular cases of semantic mismatch related to name clashing and/or misnaming.

Possibility of test reduction: can we trust features of classes we inherit from, or should we re-test derived classes from scratch? An optimistic view claims that only little or even no test is needed for classes derived from thoroughly tested classes [4]. A deeper and more realistic approach argues that methods of derived classes need to be re-tested in the new context [16]. An inherited method can behave erroneously due to either the derived class having redefined members in an inappropriate way or the method itself invoking a method redefined in the subclass.

Re-use of test cases: it is possible to use the same test cases generated for the base class during the testing of the derived class? If this is not possible, can we at least find a way of partially reusing such test cases? [9, 10]

Inheritance correctness: can we test whether the inheritance is truly expressing an IS-A relationship or are we just in presence of code reuse? Is this kind of test meaningful in absence of specifications for the class(es) under test? This issue is in some way related to the misleading interpretation of inheritance as a way of both reusing code and defining subtypes.

Testing of abstract classes: abstract classes can not be instantiated and thus can not be thoroughly tested. Only classes derived from abstract classes can be actually tested, but errors can be present also in the super (abstract) class

Figure 6 shows an example of inheritance. Questions which may arise looking at the example are whether should we retest the method `Circle::moveTo()` and whether is it possible to reuse test sets created for the class `Shape` to test the class `Circle`.

There are different approaches proposed in literature for coping with the testing problems introduced by inheritance. A first, simplistic approach starts from the idea that inherited code needs only minimal testing. Techniques addressing the problem from a more theoretical point of view have also been proposed and they can be divided

```
class Shape {                       class Circle : public Shape {
  private:                            private:
    Point referencePoint;               int radius;
  public:                             public:
    void erase();                       erase();
    virtual float area()=0;             float area();
    void moveTo(Point p);             ...
  ...                               };
};
```

Figure 6: An example of inheritance

in two main classes: Approaches based on the flattening of classes, and approaches based on incremental testing.

The former ones are based on the testing of subclasses as if every inherited feature had been defined in the subclass itself (i.e., flattening the hierarchy tree). The advantage of this kind of approach is related to the possibility of reusing test cases previously defined for the superclass(es) for the testing of the flattened subclass (obviously, adding new test cases when needed due to the addition of new features to the subclass). Redundancy is the price to be paid when following such approach. All features are re-tested in any case without any further consideration.

The latter ones start from the idea that both re-testing all inherited features and not re-testing any inherited features are wrong approaches for opposite reasons. Only a subset of inherited features needs to be re-tested in the new context of the subclass. The approaches differ in the way this subset is identified.

In the following we briefly describe the most relevat works developed on this topic. [11] states that methods provided by a parent class (which has already been tested) do not require heavy testing. This viewpoint is usually shared by practitioners, committed more to the quick results than to theoretical foundations.

An algorithm for selecting the methods that need to be re-tested in subclasses is presented in [4]. A similar, but more rigorous approach is presented in [16]. This latter paper starts defining a taxonomy of the kinds of attribute in a derived class depending both on whether they are added, or redefined, or left untouched and on the kind of dependencies with other methods belonging to the same class; the approach presented is based on the definition of a testing history for each class under test, on the construction of a call graph to represent intra/inter class interactions, and on an algorithm allowing, starting from the testing history and from the graph, for identifying which attributes have to be re-tested, which test cases can be re-used, and which attributes require new test cases.

Alternative approaches are based on the flattening of the class structure. [12] presents a technique that besides flattening the class structure, re-uses specifications of the parent classes. Another flattening oriented approach is presented in [29], this paper presents a technique based on the flattening of subclasses performed avoiding to test "unaffected" methods (methods not redefined and neither invoking redefined methods, nor using redefined attributes).

Finally, another approach for testing in the presence of inheritance is presented in [21]. In this case the goal is different from the ones of the previously presented papers. The focus of the work is on the testing of the consistency of the hierarchical relationships based on class' histories.

10 Conclusions

Although there is an increasing attention to the problematic of OOT, there are still many topics far to be covered. The main research effort is focused on the definition of object-oriented notations and methodologies, while little attention is paid on the testing of object-oriented systems. Unfortunately, object-orientation is not the silver bullet allowing designers and developers to produce error-free software, and good specification and design techniques do not eliminate the need for testing.

Object-oriented systems need to be tested just as much as (when not more than) systems developed using traditional procedural languages. Object-orientation peculiarities introduce new kind of problems as far as testing is concerned, and thus new ad-hoc testing techniques need to be defined (or at least traditional testing techniques need to be adapted). Moreover, concepts like adequacy and coverage, must be revisited to adapt them to the new paradigm represented by object-orientation.

So far, literature on object-oriented testing is still limited and there are still many aspects only uncomplitely addressed when not totally neglected. Aspects like information hiding, inheritance, polymorphism, and dynamic binding need to be specifically addressed and an appropriate error taxonomy must be defined, which allows for designing appropriate test cases. Traditional error taxonomies do not cope with brand new errors introduced by object-oriented features. Besides, there is only poor statistical information on which are the most frequent errors and on costs of error detection and removal.

References

[1] S. Barbey, M. Ammann, and A. Strohmeier. Open issues in testing Object Oriented software. In K. F. (Ed.), editor, *ECSQ '94 (European Conference on Software Quality)*, pages 257–267, vdf Hochschulverlag AG an der ETH Zürich, Basel, Switzerland, October 1994. Also available as Technical Report (EPFL-DI-LGL No 94/45).

[2] G. Booch. *Object Oriented Design*. Benjamin/Cummings Publ., USA, 1991.

[3] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language User Guide*. Addison-Wesley, 1997.

[4] T. J. Cheatham and L. Mellinger. Testing Object-Oriented Software Systems. In *Proceedings of the Eighteenth Annual Computer Science Conference*, pages 161–165. ACM, Feb. 1990.

[5] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Prentice Hall, London, 2 edition, 1991.

[6] A. Coen-Prosini, L. Lavazza, and R. Zicari. Assuring type safety of object-oriented languages. *Journal of Object-Oriented Programming*, 5(9):25–30, February 1994.

[7] D. Coleman, F. Hyes, and S. Bear. Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Transactions on Software Engineering*, 18(1):9–18, January 1992.

[8] W. Cook. A proposal for making eiffel type-safe. *The Computer Journal*, 32(4), 1989.

[9] R. Doong and P. Frankl. The astoot approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, April 1994.

[10] R.-K. Doong and P. G. Frankl. Case Studies on Testing Object-Oriented Programs. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*, pages 165–177, Victoria, CDN, Oct. 1991. ACM SIGSOFT, acm press.

[11] S. P. Fiedler. Object-Oriented Unit Testing. *HP Journal*, 40(3):69–74, April 1989.

[12] R. Fletcher and A. S. M. Sajeev. A framework for testing object-oriented software using formal specifications. In A. Strohmeier, editor, *Reliable Software Technologies*, number 1088 in Lecture Notes in Computer Science, pages 159–170. Springer, 1996.

[13] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, Englewood Cliffs N.J., 1991.

[14] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, June 1975.

[15] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Inc., 1996.

[16] 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 (ICSE'92)*, pages 68–80, Melbourne/Australia, May 1992.

[17] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *2nd ACM-SIGSOFT Symposium on the foundations of software engineering*, pages 154–163. ACM-SIGSOFT, December 1994.

[18] N. Hunt. Unit testing. *Journal of Object-Oriented Programming*, pages 18–23, February 1996.

[19] ISO/IEC. *International Standard ISO/IEC 8652:1995(E)*. Int. Organization for Standarization - Int. Electrotechnical Commission, 1995.

[20] R. Jones. Extended type checking in eiffel. *Journal of Object-Oriented Programming*, 5(2):59–62, 1992.

[21] S. Kirani. *Specification and Verification of Object-Oriented Programs*. PhD thesis, University of Minnesota, Minneapolis, Minnesota, December 1994.

[22] I. I. P. Ltd. Achieving testability when using ada packaging and data hiding methods. Web page, 1996. http://www.teleport.com/ qcs/p824.htm.

[23] J. McGregor and T. Korson. Testing of the polymorphic interactions of classes. Technical Report TR-94-103, Clemson University, 1994.

[24] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, N.Y., second edition, 1997.

[25] A. Orso and S. Silva. Open Issues and Research Directions in Object-Oriented Testing. Technical Report 98.5, Politecnico di Milano, Dipartimento di Elettronica e Informazione, February 1998.

[26] H. D. Pande and B. G. Ryder. Static type determination for c++. Technical Report LCSR-TR-197-A, Rutgers University, Lab. of Computer Science Research, October 1995.

[27] A. Paradkar. Inter-Class Testing of O-O Software in the Presence of Polymorphism. In *Proceedings of CASCON96*, Toronto, Canada, November 1996.

[28] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, 1991.

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

[30] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1994.

[31] B. Stroustrup and D. Lenkov. Run-time type identification for C++ (revised). In *Proc USENIX *C Conference*, Aug. 1992.

[32] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. In *International Conference on Software Maintenance*, pages 302–310. IEEE Society Press, September 1993.

[33] C. D. Turner and D. J. Robson. The testing of object-oriented programs. Technical Report TR-13/92, University of Durham, Durham, UK, February 1993.

[34] E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, SE-6(3):236–246, May 1980.

Corresponding author:
Alessandro Orso
Politecnico di Milano - Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci, 32
I-20133, Milan, ITALY
Tel. +(39)(2)2399 3638; Fax. +(39)(2)2399 3411
e-mail: orso@elet.polimi.it