

Reliable Objects: a Lightweight Approach Applied to Java

Jean-Marc Jézéquel* and Daniel Deveaux† and Yves Le Traon‡

Abstract

Small scale software developments need specific low cost and low overhead methods and tools to deliver quality products within tight time and budget constraints. This is particularly true of testing, because of its cost and impact on final product reliability. We propose a lightweight approach to embed tests into components, making them self testable. We also propose a method to evaluate testing efficiency, based on mutation techniques, which ultimately provides an estimation of a component's quality. This allows the software developer to consciously trade reliability for resources. Our methodology has been implemented in the Eiffel, Java, C++ and Perl languages. The Java implementation, built on top of iContract, is outlined here.

1 Introduction

Fast moving and highly reactive software developments, especially in mass-market and Internet software products, need low cost and low overhead methods and tools to deliver quality products within small time frames and tight budget constraints. It is often thought that heavyweight methods adopted in the context of coordinating multiple large groups cannot be directly applied to small-scale software developments. Still software engineers in small organizations are intensely interested in best practices, provided they can be refocused to take into account small-scale software engineering where timeliness and reactivity are the keywords.

This is particularly true of testing, because of its cost and impact on final product reliability. Classical views on testing and their associated testing models, based on the waterfall model, are not well-suited to a very dynamic, usually component based development process. The standardization of semi-formal modeling methods, such as UML, reveals this trend: testing can no longer be separated from specification/design/code stages. A low overhead test approach integrated with the component based development process must be defined with an associated testing philosophy.

In this paper, we propose a lightweight quality building method that can be implemented without the use of sophisticated and costly tools. We propose to embed tests into components, making them self testable. To build trust on a component, we propose to estimate the quality of its test sequence. So, the self-testable component carries an associated value, the level of trustability, which quantifies the quality of the unit test sequence for testing a given implementation of the component. This quantification is provided by selective mutation analysis, which has been adapted

*Irisa/CNRS, 35042 Rennes, France — e-mail : jezequel@irisa.fr

†UBS - VALORIA, 56000 Vannes, France — e-mail : Daniel.Deveaux@univ-ubs.fr

‡Irisa/Univ. Rennes, 35042 Rennes, France — e-mail : yletraon@irisa.fr

to object-oriented languages. Relying on this objective estimation of component trustability, the software developer would then be able to consciously trade reliability for resources to meet time and budget constraints.

The rest of the paper is organized as follows. Section 2 introduces the idea of self-testable components embedding their implementation along with their specification and test sequence. It also describe its implementation with Java. Section 3 defines the notion of test quality and shows how the programmer can measurably improve the quality of his tests. Related works are presented in Section 4.

2 Self-testable Classes

2.1 Specifying behavior with Contracts

Before considering running tests to check the quality of a component, there must be somewhere the knowledge of what the component is supposed to do in a given situation. This knowledge can be located into the head of the programmer (or the tester), or better, formalized with a dedicated specification language. However, for small scale software developments, it is seldom feasible to use formal specification technology because of tight time and other resource constraints. The notion of software contract has been defined to capture in a lightweight fashion mutual obligations and benefits among components. Experience tells us that simply spelling out unambiguously these contracts is a worthwhile design approach [JM97], that B. Meyer cornered the *Design by Contract* approach to software construction [Meyer92].

The *design by contract* approach prompts developers to specify precisely every consistency condition that could go wrong, and to assign explicitly the responsibility of its enforcement to either the routine caller (the client) or the routine implementation (the contractor). Along the line of abstract data type theory, a common way of specifying software contracts is to use boolean assertions called pre-and post-conditions for each service offered, as well as class invariants for defining general consistency properties. A contract carries mutual obligations and benefits: the client should only call a contractor routine in a state where the class invariant and the precondition of the routine are respected. In return, the contractor promises that when the routine returns, the work specified in the postcondition will be done, and the class invariant is still respected.

A second interest of such contracts, when they are run time checkable, is that they provide a specification against what a component implementation can be tested.

2.2 Making component self-testable

Due to the life cycle and possible evolution (through maintenance) of a software component, an organic link must be maintained between its specification, its test set and its current implementation. Our methodology is based on a integrated design and test approach for OO software components. Classes are considered as basic unit components. Test suites are defined as being an *organic* part of software OO component. Indeed, a component is triangle made of its specification (documentation, methods signature and invariant properties), one implementation and the test cases needed for testing it. To a component specified functionality is added a new feature which enables it to test itself: the component is made self-testable.

<i>Name</i>	<i>inherit from</i>	<i>inst</i>	<i>role</i>
Container	-	abstract	Base for all collection classes, define <code>count</code> , <code>isEmpty...</code>
Dispenser	Container	abstract	Containers to which new items can be added and existing items can be removed one at a time
Stack	Dispenser	concrete	The standard <i>Stack</i> structure.

Table 1: Three classes from the Pylon library (Open Source Software)

Based on this view, it is then the class implementor’s responsibility to ensure that all the embedded tests are satisfied. So, one can estimate the test quality relatively to the specification, a test sequence and a given implementation. As soon as the quality level is not reached, the test sequence must be enhanced. So when used, a self-testable component may test itself with a guaranteed level of quality. This quality level could be defined under several ways (such as classical definition-use coverage): in this paper mutation analysis [Offutt92] is proposed as a relevant way for analyzing the quality of a the test sequence. Quality measurement is thus defined based on the fault revealing power of the test sequence when systematic fault injection is performed. Once such a test quality estimate is associated to a set of functionally-equivalent components, the designer can choose the component with the best self-test ability.

2.3 Self-testable Classes in Java

The self-testable concept has been implemented in the Eiffel, Java, C++ and Perl languages. The Eiffel implementation has been presented in [TDJ99]. Since Eiffel has direct support for *Design by Contract* in the language, implementing self-testable classes in Eiffel is quite straightforward. On the other hand, the lack of standardized introspection facilities made it more difficult in several other aspects.

In this Section, we detail the Java implementation with the simplified example of a set of three classes that implements a generic stack, taken from the Pylon library [Arnaud98] (see Table 1). The complete source code of these classes as well as the self-testable classes distribution for all supported languages can be found on our web site [Deveaux99].

2.4 *Design by Contract* in Java

The Java language does not directly support the *Design by Contract* approach. We then need to implement *contract watch dogs* and also trace and assertion mechanisms. Two roads are possible: the first one is based on inherited functions that programmers call directly in their code, the second one is based on a contract definition syntax embedded in comments and uses a preprocessor that instruments the code before compilation (see Figure 1). Both approaches use the Java exception mechanism.

Whereas very simple to implement and explain, the first approach (based on inherited methods) has two main drawbacks:

- because of single inheritance Java choice, use of inheritance for contracts implementation prohibits the specialization of non instrumented classes,

<pre> /** * add() : Add an item to the dispenser */ public void add (Object element) { precondition ("writable", isWritable(), "real elem", !isVoid(element)); int old_count = count(); postcond ("keep elem", has(element), "", count() == old_count + 1); } (1) use of inherited methods </pre>	<pre> /** * add() : Add item to the dispenser * * @pre isWritable() // writable * @pre !isVoid(element) // real elem * * @post has(element) // keep elem * @post count() == count()@pre + 1 */ public void add (Object element) { } (2) preprocessor implementation </pre>
---	--

Figure 1: Possible syntaxes for contracts in Java

- more seriously: the contract calls are located *inside* methods, thus the contracts are forgotten if a method is redefined in a sub-class.

Recently, several preprocessors have been proposed to manage contracts. The example of this paper has been developed with `iContract` from Reto Kramer [Kramer99]; this tool is very simple to use, has no impact on production code (since only comments are used) and it completely implements the *Design by Contract* scheme (including inheritance of contractual obligations and OCL style `pre@` expressions corresponding to Eiffel old expressions).

2.5 How to make self-testable classes

Applying our *design for testability* approach, invariants are defined for each class, pre and post-conditions for each method. Beside, a method code can be instrumented with `check()` and `trace()` instructions to help further debugging. A standard solution in Eiffel and C++ is to use multiple inheritance for this, whereas in Java we must use interface inheritance plus delegation. Thus a Java class is made self-testable by making it implement the `SelfTestable` interface: a `'implements ubs.cls.SelfTestable'` clause is added in the declaration part, and the `test()` method is defined so as to delegate to the corresponding method in the utility class `'ubs.cls.SelfTest'`. To enable the self-testable class to run as a stand-alone program, a `main()` function can be appended to it (see Figure 2).

```

public static void main (String args[]) {
    Stack ImplementationUnderTest = new Stack() ;
    args = ImplementationUnderTest.testOptions (args) ;
    if (ImplementationUnderTest.test (args)) {
        System.exit(0);}
    else {System.exit(1);}
}

```

Figure 2: The `main()` function of a self-testable class

A class usually has several methods that should be all called and tested. In addition to standard methods, we define *testing methods*: each one frames a *testing unit* and has a goal (explained in its comment) which is to test that the implementation of a set of methods corresponds to their specifications. By convention, the testing method name begins with 'TST_'. Figure 3 shows the outline of a testing method for the class `Stack`. Usually, the test of a method consists in a simple call, since class invariant and method post-condition are sufficient *oracles* (see Section 3). To control behaviors that combine multiple method calls, we can import a `check()` function which behaves as the Eiffel `check` instruction or the C/C++ `assert` macro. The utility class `SelfTest` defines the `check()` method and a set of useful functions (`testTitle()`, `testMsg()`, ...) that support the management of tracing inside testing methods.

```

/**      TST_stack()      :  stack structure verification
 */
public void      TST_stack () {

    SelfTest.testTitle (3, "LIFO stack", "A stack of int") ;

    reset() ; // start with known state
    add (new Integer (1)) ; add (new Integer (2)) ;
    add (new Integer (3)) ;
    SelfTest.check (-1, "stack image 1", out().equals ("[1, 2, 3]")) ;
    SelfTest.testMsg ("after three 'add()' : " + out() + " ... 0k") ;
    SelfTest.check (-1, "3 on top", ((Integer) item()).intValue() == 3 ) ;
    SelfTest.testMsg ("3 on stack top ... 0k") ;
    .....
} // ----- TST_stack()

```

Figure 3: A test function for `Stack`

The testing method names are declared in an array in the test suite order; this array is used by the test launcher method, `test()`. Moreover, in each method of the class, we add the line `SelfTest.profile()` as the first statement: this allows the counting of methods calls during the test.

For V&V operations, a class is compiled through `iContract` and then through a standard Java compiler. Then the test execution is very easy, one only has to run the class. By default, the validation test is run and produces a test report as in Figure 4. Through options on the command line, it is possible to control debugging and tracing levels, and also select which testing methods are to be executed. The execution logs can be redirected to files. This instrumentation will be useful not only for the validation phase, but also for code verification and debugging. The basic test launch allows the automatic test report to be produced.

When compiling in production mode, the `iContractTM` tool can be told to tune the level of assertion checking (or even completely bypassed), and a script (`hideTest`) can be used to *hide* the testing methods as well as the `profile()` or `check()` calls in the Java source code.

2.6 Inheritance and abstract classes

As in Eiffel, `iContractTM` allows contracts to be inherited from standard classes, abstract classes and interfaces. For example the `Container` and `Dispenser` classes declare several abstract methods (`count()`, `item()`, `has()`, `add()`, `remove()`,...) in which pre and post-conditions are defined. In


```
} .....
```

3 Estimating the Quality of Tests

In our approach, it is the programmer's responsibility to produce the test cases along with, or even before, the implementation of a class. As it has been highlighted in several other papers [BG98], *Programmers love writing tests*, because it gives them immediate feedback in an incremental development setting. Still if we want to build trust on components developed this way, it is of uttermost importance to be able to associate a quality estimate to each self-test. The idea is that if the component passes its tests, this quality estimate can also be used to quantify the trust one can have in a tested component. The chosen quality criteria proposed here is the proportion of injected faults the self-test detects when faults are systematically injected into the component implementation. This estimate is, in fact, derived from the mutation testing technique, which is adapted for OO development.

3.1 Mutation testing technique for OO domain

Mutation testing is a testing technique which was first designed to create effective test data, with an important fault revealing power [OPTZ96, VM92]. It has been originally proposed in 1978 [MLS78], and consists in creating a set of faulty versions or mutants of a program with the ultimate goal of designing a test set that distinguishes the program from all its mutants. In practice, faults are modeled by a set of *mutation operators* where each operator represents a class of software faults. To create a mutant, it is sufficient to apply its associated operator to the original program.

A test set is relatively adequate if it distinguishes the original program from all its non-equivalent mutants.

Otherwise, a *mutation score*(MS) is associated to the test set to measure its effectiveness in terms of percentage of the revealed non-equivalent mutants. It is to be noted that a mutant is considered equivalent to the original program if there is no input data on which the mutant and the original program produce a different output. A benefit of the mutation score is that even if no error is found, it still measures how well the software has been tested giving the user information about the program in the absence of errors. It can be viewed in a way as a reliability assessment for the tested software.

Selective mutation aims at reducing the computational expense of mutation testing by limiting the number of mutation operators to be applied. Many expensive operators can be omitted without regression in terms of fault revealing power of the generated tests. A fault is considered from syntactic or semantic points of views. The semantic size of a mutation represents its impact on the outputs of the program while the syntactic size represents the syntactic importance of the modification. Since mutation have quite a small syntactic size (at most one instruction is modified), mutation selective operators have to find the better trade-off between semantic large or small faults. Actually, there is no way to easily appraise such semantic size, except maybe by sensitive analysis as in [VM92].

In the context of our methodology, we are looking for a subset of mutation operators general enough to be applied to various OO languages (Java, C++, Eiffel etc), implying a limited

EHF	Causes an exception when executed. This semantically large mutation operator allows to force code coverage.
AOR	Replaces occurrences of arithmetic operators by its inverse (e.g.; "+" by "-").
LOR	Each occurrence of one of the logical operators (and, or, nand, nor, xor) is replaced by each of the other operators; in addition, the expression is replaced by TRUE and FALSE.
ROR	Each occurrence of one of the relational operators (<, <=, >, >=, =, /=) is replaced by one or more of the other operators in such a way that semantically large mutations are avoided.
NOR	Replaces each statement by the empty statement.
VCP	Constant and variables values are slightly modified to emulate domain perturbation testing. Each constant or variable of arithmetic type is both incremented by one and decremented by one. Each boolean is replaced by its complement.
RFI	Stuck-at nil the reference of an object after its creation. Suppress a clone or a copy instruction. Insert a clone instruction for each reference assignment.

Table 2: Selective Mutation Operators

computational expense, and ensuring at least control-flow coverage of methods.

Our current choice of mutation operators is described in Table 2. During the test selection process, a mutant program is said to be *killed* if at least one test case detects the fault injected into the mutant. Conversely, a mutant is said to be *alive* if no test cases detects the injected fault.

3.2 Component and system test quality

The test quality of a component is simply obtained by computing the mutation score for the unit testing test suite executed with the self-test method.

The system test quality is defined as follows: let S be a system composed of n components denoted $C_i, i \in [1..n]$, let d_i be the number of killed mutants after applying the unit test sequence to C_i , and m_i the total number of mutants.

The test quality, i. e. the mutation score MS , for C_i being given a unit test sequence T_i and the *System Test Quality (STQ)* relatively to d_i and m_i , are defined by the following expressions:

$$MS(C_i, T_i) = \frac{d_i}{m_i} \quad STQ(S) = \frac{\sum_{i=1,n} d_i}{\sum_{i=1,n} m_i}$$

These quality parameters are associated to each component and the global system test quality is computed and updated depending on the number of components actually integrated into the system.

3.3 Test selection process

The whole process can be either quality driven or effort driven. In the first case, the test selection is guided by test quality, while in the second case the selection is guided by test effort constraint (estimated by a number of test cases). The whole process for generating unit test cases is divided into steps which are presented in Figure 6. Indeed, the first step concerns the mutants generation. Then, the test enhancement process is applied. It consists in applying each test case on each alive mutant. If a mutant is still alive after the execution of each test cases, then a diagnosis stage occurs, which cannot be completely automated. For each alive mutant, this stage determines why the test cases have not detected the injected fault. The diagnosis may lead to three possible actions:

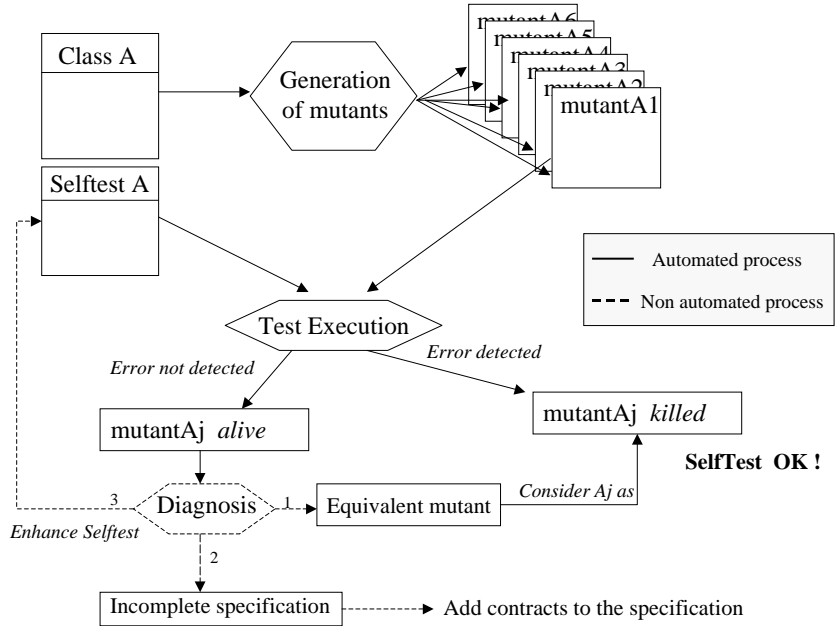


Figure 6: A Testing Process based on Mutation Analysis

equivalent mutant elimination, test cases set enhancement (tests are inadequate) or specification enhancement (specification is incomplete). This analysis is very helpful for increasing a component's quality, since it enforces the organic link between the specification/test/implementation facets of the component. After a test cases set reaches the wanted quality level, a reduction process is applied to delete redundant test cases from the set. It consists in creating the matrix marking which test cases kill which mutants. A classical matrix boolean reduction algorithm allows the final test cases set to be of minimum size with the same fault revealing power. It has to be noted that when the set of test cases is selected, the mutation score is fixed as well as the test quality of the component. Moreover, except for the diagnosis step, the process can be completely automated. The main steps of the algorithm are summarized in Figure 7.

3.4 Test cases generation and oracle determination

Deterministic test data generation is used since each class is made of a set of functionally coherent methods.

Basic efficient data are easy to generate for designers and developers. Indeed, experience teaches that deterministic test generation can follow the following rules: (1) Methods which are functionally linked belong to a same family of testing; they have to be tested together (like for a container `remove` and `add`, since you cannot test `remove` without adding one element in the container). (2) In a family, basic independent sets of methods have to be exercised first; for example, `has` cannot be tested before `add` but also you need `has` to check if `add` is correct. So `has` and `add` must be tested

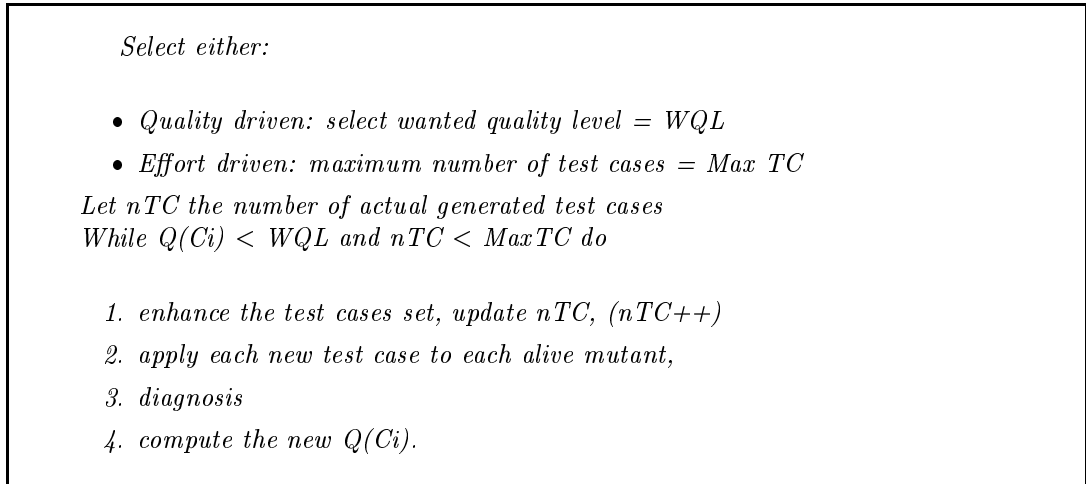


Figure 7: Algorithm for the Test Selection Process

together first, before **remove** which is less basic and needs both **has** and **add** to be tested. (3) In case of inheritance, redefined methods have to be re-tested with specific tests.

For generating oracles, the most general solution consists of writing explicit test oracles for each test suite. For example, designer knows that a `[add(2), remove(2)]` test sequence implies that `[has(2)]` should return *false*. The oracle is thus simply obtained by checking that 2 is not present in the container of integers. To be coherent with the rules expressed upward, the method `has` should have been tested before this test suite is exercised. The second way is adapted to a *design by contract* approach. In this approach, post-conditions, i.e. invariant expressions on the output domain values and relationships with the input ones, are used as partial oracle functions. Many of the faults can be detected in a systematic *design by contract* approach without writing explicit oracle functions. Post-conditions are thus covering a larger space of test data but are generally not sufficient for detecting particular semantically rich test results. In some cases, the post-condition is sufficiently precise to replace deterministic oracles: for a sort method, testing if the result is effectively sorted is a complete and simple-to-express oracle function. However, in most cases, functional dependencies between methods are difficult to express through general invariants.

4 Related works

Very few of the numerous first-generation books on analysis, design, and implementation of object-oriented software explicitly addressed V&V issues. Despite this initial lack of interest, testing of object-oriented systems is now receiving much more attention (see [Binder96] for a detailed state of the art).

Concerning OO testing techniques, most of the works focus on the dynamic aspects of OO systems: a system is viewed as a set of cooperating agents, modeling objects, and modeled with FSM, or equivalent object-state modeling [Binder94, JE94, KGJ+96]. Such works have to deal with limitations concerning computational expense of mapping objects behaviors into the underlying model. One solution consists in decomposing the program into hierarchical and functionally coherent parts. In such approaches, this decomposition provides a framework for unit, integration

and system test definition. In [MK94], the waterfall model is overtaken and an integrated test and development approach is proposed. These state-based models constrain the design methodology to divide the system into small parts with respect to behavioral complexity. Binder details the existing analogy between hardware and OO software testing and suggests an OO testing approach close to the *built-in-test* and *design-for-testability* hardware notions. In this paper, we go even further than Binder suggests, and detail how to create self-testable OO components, with an explicit analogy with the *built-in self-test* hardware terminology.

Besides, the test problem may be seen from a pragmatic point of view, and some simple-to-apply methodology can be found in the literature, which are based on an explicit test philosophy [BG98]. In this paper, the proposed methodology is based on pragmatic unit test generation. It can also serve as a basis for bridging the existing gap between unit and system dynamic tests through incremental integration testing [TJMM99]. An original measure of the quality of components has been defined based on the quality of their associated tests (itself based on fault injection). For measuring test quality, the presented approach differs from classical mutation analysis [OPTZ96, MO91] as follows: a reduced set of mutation operators is needed, oracles functions are integrated to the component, while classical mutation analysis uses differences between original program and mutant behaviors to craft a pseudo-oracle function.

5 Conclusion

The approach presented in this paper aims at providing a consistent and practical design-for-testability methodology adapted to very dynamic small-scale software developments. At a general level, our approach is quite language independent (actually it has been implemented in the Eiffel, Java, C++ and Perl languages), but on the details it must be adapted to the peculiarities of each of these languages. We have shown how it could be specialized for Java. The production of self-testable components has been detailed as well as a methodology for ensuring test quality. The approach is based on the following considerations: writing tests is easy at a unit class level [BG98], verifying the test quality is feasible through fault injection. The process of estimating test quality can be automated, and since a test driver consists in a set of self-test method calls, structural system tests are easy to launch. Relying on this objective estimation of component trustability, the software developer would then be able to consciously trade reliability for resources to meet time and budget constraints.

Further work will detail experimental studies for validating the relevance of mutation operators (both language independent and language specific) and integration strategies based on the underlying test dependency model.

References

- [Arnaud98] Arnaud (Franck). – Pylon: a foundation library. – <http://www.altsoft.demon.co.uk/free/>, 1998.
- [BG98] Beck (K.) et Gamma (E.). – Test-infected: Programmers love writing tests. *Java Report*, juillet 1998, pp. 37–50.

- [Binder94] Binder (Robert V.). – Design for testability with object-oriented systems. *Communications of the ACM*, vol. 37, n° 9, septembre 1994, pp. 87–101.
- [Binder96] Binder (Robert V.). – Testing object-oriented software : A survey. *Journal of Software Testing, Verification and Reliability*, vol. 6, n° 125-252, 1996.
- [Deveaux99] Deveaux (Daniel). – Distribution des classes auto-testables sur internet. – <http://www.iu-vannes.fr/docinfo/STclass>, mars 1999.
- [JE94] Jorgensen (Paul C.) et Erickson (Carl). – Object-oriented integration testing. *Communications of the ACM*, vol. 37, n° 9, sep 1994, pp. 30–38.
- [JM97] Jézéquel (J.-M.) et Meyer (B.). – Design by contract: The lessons of Ariane. *Computer*, vol. 30, n° 1, janvier 1997, pp. 129–130.
- [KGJ⁺96] Kung (David C.), Gao, Jerry, Chen et Cris. – On regression testing of object-oriented programs. *The Journal of Systems and Software*, vol. 32, n° 1, janvier 1996.
- [Kramer99] Kramer (Reto). – icontract(tm) and idoc(tm) distribution. – <http://www.reliable-system.com/tools>, 1999.
- [Meyer92] Meyer (B.). – Applying "design by contract". *IEEE Computer (Special Issue on Inheritance & Classification)*, vol. 25, n° 10, octobre 1992, pp. 40–52.
- [MK94] McGregor (John D.) et Korson (Tim). – Integrating object-oriented testing and development processes. *Communications of the ACM*, vol. 37, n° 9, septembre 1994, pp. 59–77.
- [MLS78] Millo (R. De), Lipton (R.) et Sayward (F.). – Hints on test data selection : Help for the practicing programmer. *IEEE Computer*, vol. 11, 1978, pp. 34–41.
- [MO91] Millo (R. De) et Offutt (A.). – Constraint-based automatic test data generation. *IEEE Transactions On Computers*, vol. 17, 1991, pp. 900–910.
- [Offutt92] Offutt (A. J.). – Investigation of the software testing coupling effect. *ACM Transaction on Software Engineering Methodology*, vol. 1, 1992, pp. 3–18.
- [OPTZ96] Offutt (J.), Pan (J.), Tewary (K.) et Zhang (T.). – An experimental evaluation of data flow and mutation testing. *Software Practice and Experience*, vol. 26, n° 2, 1996.
- [TDJ99] Traon (Yves Le), Deveaux (Daniel) et Jézéquel (Jean-Marc). – Self-testable components: from pragmatic tests to a design-for-testability methodology. In : *Proc. of TOOLS-Europe'99*. TOOLS. – To be published.
- [TJJM99] Traon (Yves Le), Jéron (Thierry), Jézéquel (Jean-Marc) et Morel (Pierre). – Efficient OO integration and regression testing. *IEEE Trans. on Reliability*, vol. To be published, n° 4, décembre 1999, pp. –.
- [VM92] Voas (J.) et Miller (K.). – The revealing power of a test case. *Software Testing, Verification and Reliability*, vol. 2, 1992, pp. 25–42.