

## Reliable Objects: Lightweight Testing for OO Languages

**Jean-Marc Jézéquel**, *Irisa, University of Rennes*

**Daniel Deveaux**, *Valoria, University of Bretagne Sud*

**Yves Le Traon**, *Irisa, University of Rennes*

**I**n the fast-moving, highly reactive arena of software development, low-cost, low-overhead methods are key to delivering high-quality products quickly and within tight budgets. Among the many aspects of the development process, testing is an obvious target for such methods, because of both its cost and its impact on product reliability. But classical views on testing and their associated test models, based on the waterfall model,

don't work well with an incremental, object-oriented development process. The standardization of semiformal modeling methods, such as UML, reveals the trend toward exactly this kind of process: We can no longer separate testing from specification, design, and coding. What we need is a testing philosophy geared toward OO development and a low-overhead test approach that we can integrate into the process.

We propose a lightweight, quality building method that developers can implement without sophisticated and costly tools. The basic idea is to embed tests into components (loosely defined here as sets of tightly coupled classes that can be deployed independently), making them self-testable. By this method, to establish a self-testable component's reliability, we estimate the quality of its test sequence. Thus, we can associate each self-testable component with a value—its level of trustability—that quanti-

fies the unit test sequence's effectiveness at testing a given implementation of the component. Our method for achieving this quantification is a version of selective mutation analysis that we adapted to OO languages. Relying on this objective estimation of component trustability, the software developer can then consciously trade reliability for resources to meet time and budget constraints. In this article, we outline the Java implementation of our methodology.

### Self-testable classes

Now let's turn to the idea of self-testable components, in which we embed a component's specification and test sequence along with its implementation into a deployment unit.

### Specifying behavior with contracts

Before we consider running tests to check a component's quality, we must know what the component should do in a given situa-

To keep test costs and overhead low, the authors propose making software components self-testable. Using mutation techniques to evaluate such components' testing efficiency gives an assessment of their quality.

tion. This knowledge might exist only in the brain of the programmer or the tester; ideally, it is formalized in a dedicated specification language. However, for many software developments, formal specification technology is seldom feasible because of tight constraints on time and other resources.

The notion of *software contracts* offers a lightweight way to capture mutual obligations and benefits among components. Experience tells us that simply spelling out these contracts unambiguously is a worthwhile design approach to software construction, for which Bertrand Meyer coined the phrase *design by contract*.<sup>1</sup>

The design-by-contract approach prompts developers to specify precisely every consistency condition that could go wrong and to assign explicitly the responsibility of each condition's enforcement to either the routine caller (the client) or the routine implementation (the contractor). Along the lines of abstract data type theory, a common way of specifying software contracts is to use Boolean assertions (called pre- and postconditions) for each service offered, along with 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 respecting the class invariant and the routine's precondition. In return, the contractor promises that when the routine returns, the work specified in the postcondition will be done and the class invariant will still be respected.

A second benefit of such contracts, when they are runtime checkable, is that they provide a specification against which we can test a component's implementation.

### Making components self-testable

Because a software component can evolve during its life cycle (through maintenance, for example), an organic link between its specification, test set, and current implementation is crucial. In our methodology, based on an integrated design and test approach for OO software components, we consider a set of tightly coupled classes as a basic unit component and define a test suite as an organic part of each component. Indeed, we think of each component as a triangle composed of its specification (documentation, methods signature, pre- and postconditions, and invariant properties), one implementation, and the test cases

**Table 1**

### Three Classes from the Pylon Library (Open Source Software)

Class name	Inherits from	Inst	Role
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	Standard stack structure

needed for testing it. To a component's specified functionality, we add a new feature that makes it self-testable.

In this approach, the class implementor must ensure that all the embedded tests are satisfied so that we can estimate test quality relative to the specification, a test sequence, and a given implementation. If the component does not reach the necessary quality level, the class implementor has to enhance the test sequence. Thus, a self-testable component can test itself with a guaranteed level of quality.

We could define this quality level several ways—for example, the classical definition involves code coverage. We propose mutation analysis<sup>2</sup> as a relevant way for analyzing a test sequence's quality. Therefore, we base the quality level on the test sequence's fault-revealing power under systematic fault injection. Once a designer can associate such test quality estimates with a set of functionally equivalent components, he or she can choose the component with the best self-testability.

### Self-testable classes in Java

We've implemented the self-testable concept in the Eiffel,<sup>3</sup> Java, C++, and Perl languages. Because Eiffel has direct support for design by contract, implementing self-testable classes in that language is straightforward. On the other hand, the lack of standardized introspection facilities made it more difficult in several other aspects.

To outline the Java implementation, we'll use the simplified example of a set of three classes taken from the Pylon library ([www.nenie.org/eiffel/pylon](http://www.nenie.org/eiffel/pylon)), implementing a generic stack (see Table 1). Our Web site ([www.iu-vannes.fr/docinfo/stclass](http://www.iu-vannes.fr/docinfo/stclass)) includes the complete source code of these classes as well as the self-testable classes distribution for all supported languages.

### Design by contract in Java

Because Java does not directly support the design-by-contract approach, we must implement contract watchdogs and trace and asser-

```

/**
 * 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() == oldcount + 1);
}

```

(a)

```

/**
 * 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)
{
    .....
}

```

(b)

**Figure 1. Possible syntaxes for contracts in Java, showing the use of (a) inherited methods and (b) the preprocessor implementation.**

```

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 Java self-testable class.**

```

/** 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() + "... Ok") ;
    SelfTest.check (-1, "3 on top," ((Integer)
item()).intValue() == 3) ;
    SelfTest.testMsg ("3 on stack top... Ok") ;
    .....
} // ----- TST_stack()

```

**Figure 3. A test function for Stack.**

tion mechanisms. (A contract watchdog monitors the contract validity on entry and exit of any method call and, in case of a contract violation, raises an exception.) To this end, two roads are possible: the first uses inherited functions that programmers call directly in their code; the second uses 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.

Although it is very simple to implement and explain, the inherited-methods approach has two major drawbacks:

- Because of Java's single inheritance, using inheritance for implementing contracts prohibits the specialization of noninstrumented classes.
- More seriously, the contract calls are located *inside* methods; thus, the contracts are forgotten if a method is redefined in a subclass.

Recently, researchers have proposed several preprocessors to manage contracts. We developed the example we use in this article with iContract from Reto Kramer ([www.reliable-systems.com/tools](http://www.reliable-systems.com/tools)). This tool is very simple to use, and, because it uses only comments, it does not affect production code. In addition, it completely implements the design-by-contract scheme, including inheritance of contractual obligations and Object Constraint Language-style @pre expressions corresponding to Eiffel old expressions.

### How to make self-testable classes

Applying our design-for-testability approach, we define invariants for each class and pre- and postconditions for each method. In addition, we can instrument a method code with check() and trace() instructions to help further debugging. A standard solution for this in Eiffel and C++ is to use multiple inheritance; in Java, we must use interface inheritance plus delegation. Thus, we make a Java class self-testable by making it implement the SelfTestable interface: We add an implements `ubs.cls.SelfTestable` clause to the declaration and define the test() method so as to delegate to the corresponding method in the `ubs.cls.SelfTest` utility class. To enable the self-testable class to run as a standalone program, we

can append a `main()` function (see Figure 2).

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

An array in the test suite order declares the testing method names; the test launcher method `test()` uses this array. Moreover, in each method of the class, we add `SelfTest.profile()` as the first statement: this allows the counting of method calls during the test.

For validation and verification, we compile a class through `iContract` and then through a standard Java compiler. At that point, test execution is very easy; we only have to run the class. By default, running the class runs the validation test and produces a test report such as that in Figure 4. Through options on the command line, we can control debugging and tracing levels and select which testing methods to execute. We can also redirect the execution logs to files. This instrumentation is useful not only for the validation phase, but also for code verification and debugging.

When compiling in production mode, we can tell the `iContract` tool to tune the level of assertion checking or even bypass it. A script called `hideTest` lets us hide the testing methods as well as the `profile()` or `check()` calls in the Java source code.

### Inheritance and abstract classes

As in Eiffel, `iContract` 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()`, and so on) that define pre-

```

....>java ubs.struct.Stack -stat

Test of class 'ubs.struct.Stack'

Test unit n. 1 Container creation

    empty, readable and writable
    - newly created Container... Ok
    - Container empty... Ok
    - Container readable... Ok
    ....
    -----

Test unit n. 3 LIFO stack

    A stack of int
    - three 'add()' : [1, 2, 3]... Ok
    - 3 on stack top ... Ok
    ....

Test TST_stack ended
Number of called methods      :   49
Number of aborted calls      :    0
-----

End of test sequence
Number of called methods      :  101
Number of aborted calls      :    0

List of tested methods
add                          :    6
count                        :   39
has                          :    6
isEmpty                      :   11
.....
    -----

```

**Figure 4. Test report of the Stack class.**

```

/** add() : Add item to dispenser
 *
 * @pre isWritable()           // writable
 * @pre ! isVoid (elem)       // real element
 *
 * @post has(elem)            // keep element
 * @post count() == count()@pre + 1
 */
public abstract void add (Object elem);

    in Dispenser.j

/** add() : PUSH
 *
 * @post elem == item() // new on top
 */
public void add (Object elem)
{
    SelfTest.profile();
    ....
}
    in Stack.j

```

**Figure 5. Postcondition strengthening in add().**

and postconditions. The concrete class `Stack` implements these methods but does not redefine the contracts. On the other hand, in other methods, a precondition might need to be weakened or a postcondition might need to be strengthened (see Figure 5).

**The quality criterion we propose here is the proportion of injected faults the self-test detects when we systematically inject faults into the component implementation.**

Because abstract classes can have neither a constructor nor a `main()` function, we cannot make them self-testable. Nevertheless, an abstract class contains testing methods inherited by subclasses. We can write these testing methods, because very often at the abstract class level, we know the way a set of methods (both abstract and concrete) should be used. In our example, `Container` defines a testing method (`TST_create()`) that tests the instantiation mechanism and the methods `count()`, `isEmpty()`, and so on. In the same way, `Dispenser` defines `TST_addsup()`, which tests `add()` and `remove()`. The concrete class `Stack` defines only one testing method (`TST_stack()`, shown in Figure 3), but it uses all three methods in its testing sequence.

It is very easy to reuse testing units, especially in large libraries that use inheritance widely. In such cases we can improve the reuse mechanism, redefining a testing method and calling a parent method:

```
public void TST_foo() {
    super.TST_foo() ;
    .....
}
```

### Estimating test quality

In our approach, it is the programmer's responsibility to produce the test cases along with, or even before, implementing a class. As several other researchers have pointed out,<sup>4</sup> programmers love writing tests because it gives them immediate feedback in an incremental development setting. Still, if we want to build trust in components developed this way, it is of the utmost importance to have a quality estimate for each self-test. If the component passes its tests, we can use this quality estimate to quantify our trust in the component.

The quality criterion we propose here is the proportion of injected faults the self-test detects when we systematically inject faults into the component implementation. We derived this criterion from mutation testing techniques adapted for OO development.

### Mutation testing for OO

Mutation testing was first developed to create effective test data with significant fault-revealing power.<sup>5,6</sup> As originally proposed by De Millo, Lipton, and Sayward in

1978,<sup>7</sup> the technique is to create 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, the technique models faults using a set of *mutation operators*, where each operator represents a class of software faults. To create a mutant, you apply its operator to the original program.

A test set is relatively adequate if it distinguishes the original program from all its nonequivalent mutants. Each test set receives a *mutation score* that measures its effectiveness in terms of the percentage of revealed nonequivalent mutants. (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 the test set finds no errors, the score still measures how well the software has been tested. This gives the user information about certain kinds of errors being absent from the program. Thus, the technique provides a kind of reliability assessment for the tested software.

Selective mutation reduces the computational expense of mutation testing by limiting the number of mutation operators applied. (We can omit many expensive operators without losing anything in terms of the tests' fault-revealing power.) Selective mutation considers faults from syntactic and semantic points of view. A mutation's semantic size represents its impact on the program's outputs; the syntactic size represents the modification's syntactic importance. Because mutations have quite a small syntactic size (modifying one instruction at most), selective-mutation operators must find the better trade-off between large and small semantic faults. Actually, there is no easy way to appraise semantic size, except possibly by sensitive analysis.<sup>6</sup>

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, and so on), implying limited computational expense, and ensuring at least control-flow coverage of methods. Table 2 summarizes our current choice of mutation operators.

During the test selection process, we say that a mutant program is *killed* if at least one test case detects the fault injected into the

mutant. Conversely, we say that a mutant is *alive* if no test case detects the injected fault.

### Component and system test quality

We obtain a component's test quality simply by computing the mutation score for the unit-testing test suite executed with the self-test method.

We define the system test quality 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 let  $m_i$  be the total number of mutants. To define test quality—that is, the mutation score  $MS$ , for  $C_i$  being given a unit test sequence  $T_i$ —and system test quality ( $STQ$ ) relative to  $d_i$  and  $m_i$ , we use the following expressions:

$$MS(C_i, T_i) = d_i/m_i$$

$$STQ(S) = \hat{A}_{i=1,n} d_i / \hat{A}_{i=1,n} m_i.$$

We associate these quality parameters with each component and compute and update the global system's test quality depending on the number of components actually integrated into the system.

### Test selection process

The whole process can be driven by either quality or effort. In the first case, test quality guides test selection, while in the second case, some test effort constraint (estimated by a number of test cases) guides the selection.

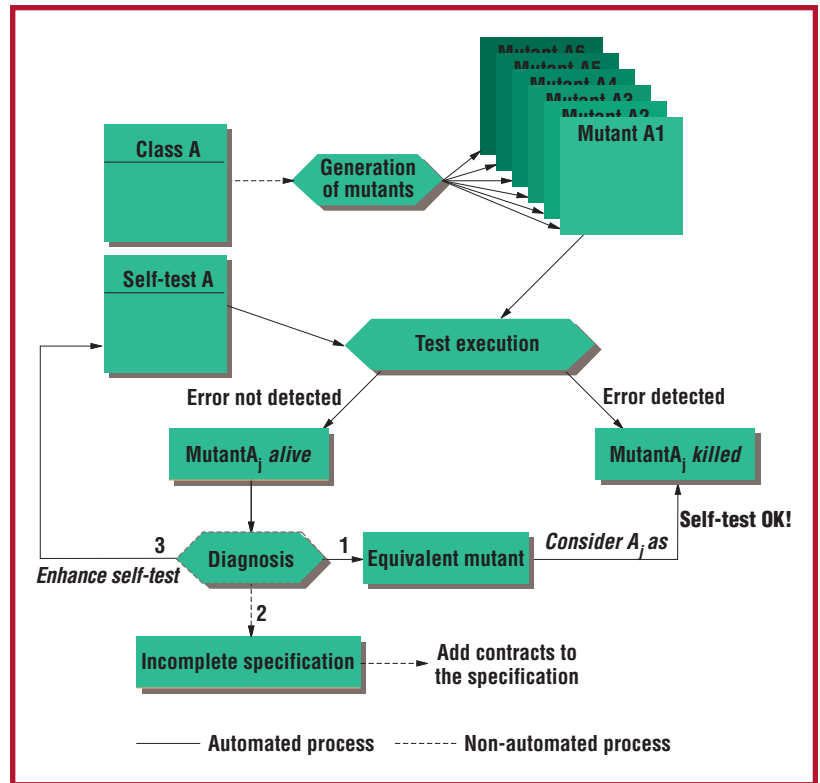
Figure 6 outlines the process of generating unit test cases. The first step is mutant generation. Next comes the test enhancement process, which consists of applying each test case to each live mutant. If a mutant is still alive after execution of each test case, then a diagnosis stage occurs, which cannot be completely automated. For each live mutant, diagnosis determines why the test cases have not detected the injected fault. The diagnosis leads to one of three possible actions: equivalent mutant elimination, test-case set enhancement (if the tests are inadequate), or specification enhancement (if the specification is incomplete). This analysis is very helpful for increasing a component's quality, because it enforces the organic link between the component's specification, test, and implementation facets.

After a test-case set reaches the desired

**Table 2**

### Selective-Mutation Operators

Operator	Action
EHF	Causes an exception when executed. This semantically large mutation operator allows forced code coverage.
AOR	Replaces occurrences of arithmetic operators with their inverses (for example, + replaces -).
LOR	Replaces each occurrence of a logical operator (and, or, nand, nor, xor) with each of the other operators. Also replaces the expression TRUE with FALSE.
ROR	Replaces each occurrence of one of the relational operators (<, >, ≤, ≥, =, ≠) with one or more of the other operators in a way that avoids semantically large mutations.
NOR	Replaces each statement with the empty statement.
VCP	Slightly modifies constant and variable values 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	Resets an object's reference to the null value. Suppresses a clone or a copy instruction. Inserts a clone instruction for each reference assignment.



**Figure 6. Testing process based on mutation analysis.**

quality level, we apply a reduction process to delete redundant test cases. This process consists of creating a matrix marking which test cases kill which mutants. A classical matrix Boolean reduction algorithm generates the final test-case set, minimizing the set size while maintaining the fault-revealing power. Selection of the test-case set fixes the mutation score as well as the component's test quality. Except for the diagnosis

```

Select one of the following:
  Quality-driven: Wanted quality level = WQL
  Effort-driven: Maximum number of test cases = MaxTC

Let nTC be the number of actual generated test cases.
While  $Q(Ci) < WQL$  and  $nTC < MaxTC$ , do

  1. Enhance the test-case set, update nTC, (nTC++);
  2. Apply each new test case to each live mutant;
  3. Diagnosis;
  4. Compute the new  $Q(Ci)$ .

```

**Figure 7. Algorithm for the test selection process.**

step, the process can be completely automated. Figure 7 summarizes the algorithm's main steps.

### Test case generation and oracle determination

Our technique uses deterministic test data generation because each class consists of a set of functionally coherent methods. Designers and developers can easily generate basic efficient data; experience teaches that deterministic test generation can follow these rules:

- Methods that are functionally linked belong to the same testing family; they must be tested together. (For example, a container `remove` and `add` must be tested together because you cannot test `remove` without adding one element in the container.)
- In a family, basic independent sets of methods must be exercised first; for example, `has` cannot be tested before `add`, but you also need `has` to check that `add` is correct. So `has` and `add` must be tested together first, before `remove`, which is less basic and needs both `has` and `add` to be tested.
- In case of inheritance, redefined methods must be retested with specific tests.

To generate oracles, the most general solution is to write explicit test oracles for each test suite. For example, designers know that an `[add(2), remove(2)]` test sequence implies that `[has(2)]` should return *false*. We thus obtain the oracle simply by checking that 2 is not present in the container of integers. To be coherent with the rules expressed upward, the method should have been tested before the test suite is exercised.


The second way to generate oracles works well with design by contract. In this approach, we use postconditions—invariant expressions on the output domain values

and relationships with the input ones—as partial oracle functions. Thus, in a systematic design-by-contract approach we can detect many of the faults without writing explicit oracle functions.

Postconditions cover a larger test-data space than the explicit test oracles but are generally not sufficient for detecting semantically rich test results. In some cases, a postcondition is sufficiently precise to replace deterministic oracles: For a sort method, testing whether 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.

**W**e hope the approach we've presented here will provide a consistent, practical design-for-testability methodology for the growing number of software development projects for which timeliness and reactivity are key. The key benefits of our approach are these:

- Writing tests is easy at the unit class level.<sup>4</sup>
- Verifying the test quality is feasible through fault injection.
- The process of estimating test quality can be automated.
- Because a test driver consists of a set of self-test method calls, structural system tests are easy to launch.
- Relying on the objective estimation of component trustability that our method provides, software developers can consciously trade reliability for resources to meet time and budget constraints.

For future work, we've planned experimental studies for validating the relevance of both language-independent and language-specific mutation operators. We'll also look at integration strategies based on the underlying test dependency model. 

## References

1. B. Meyer, "Applying 'Design by Contract,'" *Computer*, vol. 25, no. 10, Oct. 1992, pp. 40–52.
2. J. Offutt, "Investigation of the Software Testing Coupling Effect," *ACM Trans. Software Engineering Methodology*, vol. 1, no. 1, Jan. 1992, pp. 5–20.
3. Y. Le Traon, D. Deveaux, and J.-M. Jézéquel, "Self-Testable Components: From Pragmatic Tests to a Design-for-Testability Methodology," *Proc. TOOLS-Europe'99*, IEEE Computer Society, Los Alamitos, Calif., 1999, pp. 96–107.
4. K. Beck and E. Gamma, "Test-Infected: Programmers Love Writing Tests," *Java Report*, vol. 3, no. 7, Jul. 1998, pp. 37–50.
5. J. Offutt et al., "An Experimental Evaluation of Data Flow and Mutation Testing," *Software Practice and Experience*, vol. 26, no. 2, Feb. 1996, pp. 165–176.
6. J. Voas and K. Miller, "The Revealing Power of a Test Case," *Software Testing, Verification and Reliability*, vol. 2, no. 1, May 1992, pp. 25–42.
7. R. De Millo, R. Lipton, and F. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol. 11, no. 4, Apr. 1978, pp. 34–41.

## About the Authors



**Jean-Marc Jézéquel** is a professor at IRISA, University of Rennes, France, where he leads an INRIA research project called Triskell. His research interests include software engineering based on OO technologies for telecommunications and distributed systems. He received an engineering degree in telecommunications from the École Nationale Supérieure des Télécommunications de Bretagne and a PhD in computer science from the University of Rennes. Contact him at IRISA, Campus de Beaulieu, F-35042 RENNES (France); jezequel@irisa.fr.

**Daniel Deveaux** is an assistant professor at the University of Bretagne Sud, where he leads the Aglae research team. His research interests include OO design and programming, with an emphasis on documentation, testing, and trusted software component production methodologies. He received an engineering degree in agronomy from the École Nationale Supérieure d'Agronomie de Rennes. Contact him at UBS—Lab VALORIA, BP 561 - 56017 Vannes (France); daniel.deveaux@univ-ubs.fr.



**Yves Le Traon** is an assistant professor at IRISA, University of Rennes, France. His research interests include testing, design for testability, and software measurement. He received his engineering degree and his PhD in computer science from the Institut National Polytechnique de Grenoble, France. Contact him at IRISA, Campus de Beaulieu, F-35042 RENNES (France); yletraon@irisa.fr.

Few of the numerous first-generation books on analysis, design, and implementation of OO software explicitly address validation and verification issues. Despite this initial lack of interest, researchers have begun to devote more interest to testing of OO systems. Robert V. Binder has published a detailed state of the art.<sup>1</sup>

Concerning OO testing techniques, most researchers have focused on the dynamic aspects of OO systems: They view a system as a set of cooperating agents, modeling objects. And they model the system with finite-state machines or equivalent object-state modeling.<sup>1–3</sup> Such approaches must deal with limitations concerning the computational expense of mapping objects' behavior into the underlying model. One solution consists of decomposing the program into hierarchical and functionally coherent parts. This decomposition provides a framework for unit, integration, and system test definition. John D. McGregor and Tim Korson leave behind the waterfall model, proposing an integrated test and development approach.<sup>4</sup> However, these state-based models constrain the design methodology to dividing the system into small parts with respect to behavioral complexity.

Other researchers, approaching the test problem from a pragmatic point of view, have come up with simple-to-apply methodologies based on explicit test philosophies. Ivar Jacobson and colleagues describe how to codesign accompanying test classes with "normal" classes.<sup>5</sup> Ken Beck and Erich Gamma propose a methodology based on pragmatic unit test generation.<sup>6</sup> This methodology can also serve as a basis for bridging the existing gap between unit and system dynamic tests through incremental integration testing.<sup>7</sup>

Binder discusses the existing analogy between hardware and OO software testing and suggests an OO testing approach close to hardware notions of *built-in test* and *design for testability*. In this article, we go even further by detailing how to create self-testable OO components, explicitly using concepts and terminology from hardware's *built-in self-test*.

We also define an original measure of a component's quality based on the quality of its associated tests (in turn based on fault injection). For measuring test quality, our approach differs from classical mutation analysis<sup>8</sup> in that it uses a reduced set of mutation operators and integrates oracle functions into the component. Classical mutation analysis uses differences between the original program and mutant behaviors to craft a pseudo-oracle function.

## References

1. R.V. Binder, "Testing Object-Oriented Software: A Survey," *J. Software Testing, Verification and Reliability*, vol. 6, no. 3–4, Sept.–Dec. 1996, pp. 125–252.
2. P.C. Jorgensen and C. Erickson, "Object-Oriented Integration Testing," *Comm. ACM*, vol. 37, no. 9, Sept. 1994, pp. 30–38.
3. D.C. Kung et al., "On Regression Testing of Object-Oriented Programs," *J. Systems and Software*, vol. 32, no. 1, Jan. 1996, pp. 232–244.
4. J.D. McGregor and T. Korson, "Integrating Object-Oriented Testing and Development Processes," *Comm. ACM*, vol. 37, no. 9, Sept. 1994, pp. 59–77.
5. I. Jacobson et al., *Object-Oriented Software Engineering—A Use Case Driven Approach*, Addison-Wesley/ACM Press, Reading, Mass., 1992.
6. K. Beck and E. Gamma, "Test-Infected: Programmers Love Writing Tests," *Java Report*, vol. 3, no. 7, Jul. 1998, pp. 37–50.
7. Y. Le Traon et al., "Efficient OO Integration and Regression Testing," *IEEE Trans. Reliability*, vol. 49, no. 1, Mar. 2000, pp. 12–25.
8. J. Offutt et al., "An Experimental Evaluation of Data Flow and Mutation Testing," *Software Practice and Experience*, vol. 26, no. 2, Feb. 1996, pp. 165–176.