JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2005

Vol. 4, No. 7, September–October 2005

E-Tester: a Contract-Aware and Agent-Based Unit Testing Framework for Eiffel

Jonathan S. Ostroff, Department of Computer Science, York University, Canada.
Richard F. Paige, Department of Computer Science, University of York, U.K.
David Makalsky, Department of Computer Science, York University, Canada.
Phillip J. Brooke, School of Computing, Communications, and Electronics, University of Plymouth, U.K.

We describe a contract-aware unit testing framework, E-Tester, for the Eiffel programming language. The framework differs from JUnit in its first-class support for lightweight formal methods, through test support for contracts and assertions. As well, it supports a form of negative test, called *violation cases*, which aim at validating contracts. It also differs based on its use of *agents* for expressing tests and test cases. We compare E-Tester with JUnit and suggest several advantages it offers, with the additional aim of making recommendations for improving JUnit's support for testing software with contracts. We also explain how it can be applied within a test-driven process for building reliable systems.

1 INTRODUCTION AND MOTIVATION

Test-driven Development (TDD) [3] is one of the popular evolving agile methods [9]. Like all agile methods, TDD stresses the development of working code over documentation, models and plans. The suggestion of TDD is (paradoxically) that developers test the program before it is fully written. A striking aspect of this approach is the idea that the code that is implemented may not be behaviourally correct: it just has to pass the test. The test is therefore the specification against which the code is checked. *Automated tools*, which will make writing tests easy and running them against the code automatic, are essential for supporting TDD. Such tools now exist (e.g., JUnit for Java [7]), and have seen widespread adoption and use.

The Eiffel language [12] differs substantially from Java. One key difference is that Eiffel provides built-in support for lightweight formal methods for expressing and checking *contracts*; non-standard extensions and new languages have been proposed to add these features to Java [8, 10]. Contracts – expressed as preconditions and postconditions on methods, and invariants on classes – are used to document functionality, for debugging, and to constrain maintenance and extensions that may take place in the future. In practice, the use of contracts during development can

Cite this article as follows: Jonathan S. Ostroff, Richard F. Paige, David Makalsky, Phillip J. Brooke: "E-Tester: a Contract-Aware and Agent-Based Unit Testing Framework for Eiffel", in *Journal of Object Technology*, vol. 4, no. 7, September–October 2005, pp. 97–114, http://www.jot.fm/issues/issues_2005_09/article4

help improve the reliability and robustness of software, provided that suitable tool support for monitoring, debugging, activating, and testing contracts is provided. It is this last point that we focus on in this paper. JUnit, as we shall see, is limited in its support for testing contracts, but by examining a unit testing framework for Eiffel, we can obtain guidance on how to improve JUnit's contract support.

We describe a unit testing framework for Eiffel, called E-Tester. E-Tester has a number of similarities to JUnit but differs in its approach because of its treatment of contracts as first-class citizens. As such, it offers a different design, and provides somewhat different functionality, to JUnit. A key element in the design of E-Tester is its distinction between boolean tests (which are used to check properties against the system, as well as for checking correctness) and violation tests (which are used to validate contracts, i.e., to ensure that contracts express the properties that are desired of objects and methods). The latter in particular are used to exercise contracts, in order to ensure that the contracts capture the right properties. E-Tester is also distinguished by its use of Eiffel's *agent* technology for expressing tests.

We emphasise that E-Tester is not a test generation framework; there is parallel work at ETH Zurich on a test wizard for just this purpose [2]. E-Tester should thus be viewed as complementary to this work.

We commence with a very brief overview of Design-by-Contract, the lightweight formal method that E-Tester aims to support. We then give a short overview of agents in Eiffel, which are used within E-Tester for managing and encoding unit tests, and also for encoding quantifiers in formal specifications in Eiffel code. We next discuss JUnit and its design goals, before considering E-Tester's design goals and its overall design, followed by a comparison with JUnit – this in turn leads to suggestions on how to improve JUnit's support for testing contracts. We sketch a brief example, and then explain ongoing work on integrating E-Tester with an Eiffel plug-in for Eclipse.

2 EIFFEL AND DESIGN-BY-CONTRACT

Eiffel is an object-oriented programming language and method [12]; it provides constructs typical of the object-oriented paradigm, including classes, objects, inheritance, associations, composite ("expanded") types, generic types, polymorphism and dynamic binding, and automatic memory management.

The BON modelling language [18] is a language designed to work seamlessly and reversibly with Eiffel. We use BON in a number of examples in later sections; it offers features similar to UML's class and collaboration diagrams, though its integrated support for design-by-contract is superior to that of UML and OCL [15].

2.1 Design-by-Contract

Design-by-Contract (DbC) is a lightweight formal technique for engineering software systems with significant requirements for reliability and robustness. It integrates mathematical descriptions with code, ensuring consistency, and it is designed to be supported by tools that are comfortable and familiar to developers, e.g., compilers, debuggers, static checkers, and testing frameworks.

DbC suggests annotating classes with preconditions, postconditions, and class invariants. These assertions infer contracts that bind callers of class services with implementers of said services: callers guarantee to satisfy preconditions, while implementers guarantee to satisfy postconditions. This convention guarantees that conditions which may affect the correct operation of a class are checked only once. In Eiffel, these assertions are built in to the programming language, and the assorted Eiffel compilers and integrated development environments (e.g., ISE Eiffel and GNU SmartEiffel) provide tools for managing assertions.

In Eiffel, contracts are code: they are executable, are integrated with program constructs, and are evaluated at run-time. Because they are code, they need to be tested. Contracts may be incorrectly or incompletely specified: an Eiffel program with contracts can result in errors due to incorrectly implemented functionality, or due to improperly stated contracts. Some omissions or errors will be caught simply by executing the code and observing that there is an inconsistency between the code and the contracts. But contracts that are too weak (i.e., do not completely capture all constraints that are relevant to a method or class), or too strong (e.g., that capture properties outside of the requirements for a method or class) need not be caught simply by executing the code. As well, some conditions are difficult to express using executable contracts – the paper [14] considers examples – and it may be more convenient to use a variety of test case called a *collaborative test* (discussed in the sequel). Thus, contracts must be tested: test cases must be written that invoke methods with invalid preconditions, or generate objects such that class invariants are not satisfied. These test cases can help programmers determine whether or not their contracts are valid, i.e., whether they capture the properties that are required.

2.2 Agents

A key Eiffel technology used in E-Tester is that of *agents*. Agents are objects that represent operations; they are effectively closures from functional programming. Agents can be passed to different software elements, which can use the object to execute the operation *whenever* they want. Agents thus provide a way of separating the definition of a routine from its execution. Agents are also a way of combining high-level functions (operations acting on other operations) with static typing in Eiffel.

Here is a simple example of an agent, using Eiffel's GUI library EiffelVision. Suppose you want to add the routine *eval_state* to the list of event handlers that

will be executed when a mouse click occurs on the widget *my_button*. To carry this out, the following Eiffel statement would be executed.

$$my\_button.click\_actions.extend(\textbf{agent}\ eval\_state)$$

The operation being added to the button is indicated by the **agent** keyword. The keyword distinguishes an operation call to *eval_state* from a binding of the operation to the button. In general, the argument to *extend* can be any agent expression. An agent expression will include an operation plus any context that the operation may need (e.g., arguments).

Predicate agents are of significant use. These agents apply boolean-valued operations to collections. For example:

$$intlist.for\_all(\textbf{agent}\ is\_positive(?)) \quad (1)$$
$$intlist.there\_exists(\textbf{agent}\ perfect\_cube(?)) \quad (2)$$

The first example applies the boolean-valued function *is_positive* to elements of the integer list *intlist*, and conjoins together the result. The question mark ? indicates an open argument that is provided by iterating through the range arguments provided, i.e., it indicates an arbitrary element of *intlist*. Equation (2) applies the boolean-valued function *perfect_cube* to elements of the integer list *intlist* and disjoins the result. The question mark indicates an open argument that is provided by the list interator, i.e., it indicates some element of the integer list. Suppose, for example, we have a class *CITIZEN* which has attributes representing the citizen's set of parents and the citizen's set of children. To specify, using agents, that children and parents are linked via references, we would use an invariant clause as shown in Fig. 1.

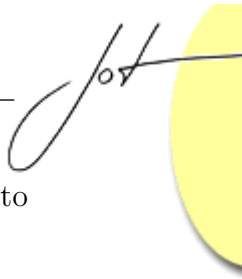
```
children.for_all((c:CITIZEN):BOOLEAN do Result := c.parents.has(Current) end)
```

Fig. 1: BON invariant clause expressed using Eiffel agents

The example in Fig. 1 illustrates anonymous agents, which are akin to inline functions in other languages; the body of the iterator `for_all` is an anonymous agent. *c*, the bound variable for the `for_all` agent, is an element taken from the collection *children*, to which the body of the anonymous operation (contained within the inner `do..end` block) is applied.

3 JUNIT

JUnit is an open source Java testing framework used to write and run repeatable tests [7]. JUnit supports assertions for testing expected results, test fixtures for sharing common test data, test suites for easily organizing and running tests, and graphical and textual test execution engines. Goals of JUnit are:



1. To require minimal work to write new tests, using familiar tools, so as to encourage developers to actually write tests.

2. To create tests that retain their value over time, by making tests usable by developers other than their author.

3. To make it possible to build new tests by leveraging existing tests, i.e., to enable reuse of fixtures.

The architecture of JUnit is shown in Fig. 2. The heart of JUnit is the class `TestCase`. A concrete test case (i.e., a class in which a developer writes tests and that extends `TestCase`) has methods that implement individual tests as well as optional `setUp` and `tearDown` methods for fixtures.

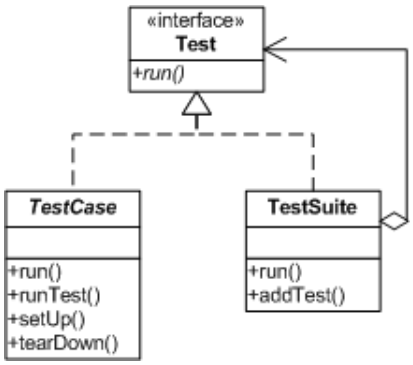


Fig. 2: JUnit Architecture

JUnit distinguishes between *failures* and *errors*. The possibility of a failure is anticipated and checked for with assertions, and is signalled with an assertion fail message. Errors, on the other hand, are unanticipated problems such as array index out of bound. Different `catch` clauses in the `run` method (see Fig. 3) distinguish between failures and errors, and ensure that all tests run to completion. A variety of assert methods are provided for failure cases, e.g., AssertTrue, AssertFalse.

Method `runTest` uses Java reflection to retrieve all methods in the subclass of *TestCase* that start with `test`, and invokes each test method.

Java does not support contracts, although from Version 1.4 and onwards the primitive `assert` mechanism has been used for basic contracting. This mechanism is nowhere near as powerful as contracting techniques offered by Eiffel, iContract [8], or JML [10], and it is not generally well integrated with debuggers and IDEs. There are several strategies that can be used to test Java code that includes primitive contracts with JUnit. One is as follows. Suppose we have

```
public class Account {
  double balance;
```

```
public void run() {
  setUp();
  try {
    runTest();
  }
  catch ... //assertion failures
  catch ... //all other unanticipated exceptions
  finally {
    tearDown();
  }
}
```

Fig. 3: The run method that implements the JUnit testing process

```
  public void withdraw(double amount) {
    assert (amount > 0);
    balance = balance - amount
  }
}
```

To test the precondition `assert (amount > 0)` we could use `fail` wrapped in a `try` block, as follows.

```
public void testAccountWithdraw() {
  Account a = new Account();
  a.balance = 20;
  try {
    a.withdraw(-30);
    fail("Should not get here");
  }
  catch(AssertException ex) {}
}
```

The call to `withdraw` will generate an assertion failure which will be caught by the exception handler. This is not a convenient structure to use for testing preconditions (and postconditions). In general, we claim that JUnit is awkward for testing expected and legitimate contract failures; thus, a different design will be required for testing a contract-aware language like Eiffel, or for a unit testing framework that integrates with a Java contract package like iContract [8] (we point out that JUnit as currently defined provides no special support for testing contracts).

A more detailed comparison of JUnit with E-Tester appears in Section 4.4.

4 E-TESTER

E-Tester is a unit testing framework for Eiffel. It consists of three clusters (packages) which, when added to an Eiffel system, will allow for straightforward development of test suites. It has the same general design goals as JUnit (see Section 3) but is tailored for Eiffel, thus providing better support for testing of systems that include lightweight formal methods represented using contracts.

E-Tester has been designed to work with ISE EiffelStudio, and makes use of the meta-referential facilities provided with this IDE. E-Tester provides its own GUI front-end, but this is not required in order to use the testing framework. It has many similarities to JUnit, and differs substantially from other existing unit testing packages for Eiffel, such as *getest* [4] and EiffelUnit (see Section 5).

4.1 Design goals for E-Tester

As stated, the general design goals for E-Tester are identical to JUnit: to allow developers to write tests using convenient and familiar tools (in this case, Eiffel code); to encourage reuse of tests; and to make tests of lasting value. However, there are additional goals for E-Tester because of its need to support testing of systems that include contracts. Particularly, E-Tester supports two different general types of test cases, which will appear when using any programming language that supports design-by-contract; we give concrete examples in the next subsection.

1. In a *boolean test case* we check for the existence of some functionality by invoking program features, and then check that the state of the computation satisfies some conditions. The test passes iff the conditions are true *and* all the contracts are satisfied (i.e., there are no unhandled assertion exceptions). This last point is critical: if a boolean test case succeeds (terminates without exceptions) then all contracts invoked during the test have also succeeded and we have also partially checked the *correctness* of the system.

   In TDD, it is anticipated that the test will initially fail, as the code for the functionality does not exist or is incomplete. In addition, even if the code is completed, it may not satisfy all the contracts, and there may be unanticipated errors such as an array index out of bound.

   Boolean test cases are partially supported by JUnit: unless Java and JUnit are used in conjunction with a contract package like iContract, the benefits and features offered by E-Tester surpass those of JUnit for running boolean test cases, since partial correctness is not checked.

2. A *violation test case* is a test that is used to check the *validity* of contracts, i.e., that the contract completely and correctly captures its requirements. For example, consider a method that reads a field from a database; its precondition is that the database is open for read access. A violation test case will invoke

this method with the database *closed*, thus checking that the precondition is correctly and completely expressed. A violation test case is similar to a negative test: in such a test, it is anticipated that the contract will fail (on either the part of the caller or the supplier of the service) and will raise an unhandled exception; thus, violation test cases typically call a routine in a state where the routine's precondition is false, or construct an object so that a routine terminates with the class invariant false. This provides a means for determining that the routine's precondition is indeed necessary and adequate.

The reason for carrying out such a test is to validate that the contracts completely and correctly capture their requirements. Such a test passes iff there is at least one contract violation which results in an unhandled exception. This type of test is not easily supported by JUnit. The design of E-Tester, however, supports it.

4.2 Design of E-Tester

The design of E-Tester is shown in Fig. 4 using a BON class diagram; classes are represented as rounded rectangles, inheritance using single directed arrows, and associations (reference relations) using thick arrows.

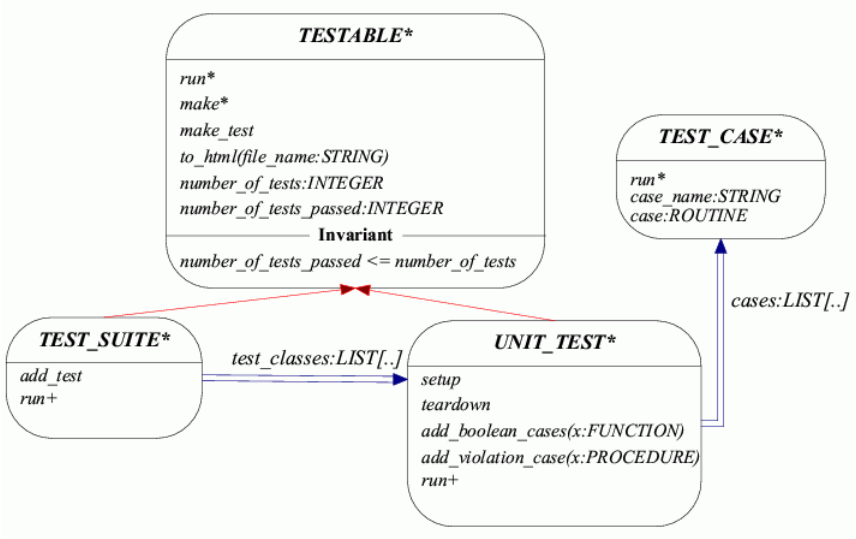


Fig. 4: Static structure of E-Tester

The design of E-Tester is somewhat different than that of JUnit, mainly because of the need to test expected contract violations, i.e., violation test cases. We illustrate this with some examples, based on Fig. 5, below.

We want to distinguish between at least the two kinds of tests discussed earlier.

Fig. 5 demonstrates the two kinds of tests discussed in Section 4.1.

```
class ACCOUNT_TEST inherit
    UNIT_TEST
create
    make  -- name of constructor

feature{NONE}
    make is  -- constructor
        do
            make_test;
            add_boolean_case (agent test_withdraw);
            add_violation_case (agent test_negative_amount);
            to_html("tests.htm")
        end

feature -- cases
    test_withdraw:BOOLEAN is   -- function
        local a: ACCOUNT
        do
            comment("test_withdraw");
            create a.make(100);
            a.withdrawal_payout(80);
            Result := a.balance = 20  ◄
        end

    test_negative_amount is  -- procedure
        local a: ACCOUNT
        do
            comment("test_negative_amount");
            create a.make(100);
            a.withdrawal_payout(-10) ◄
        end
end
```

PASSED (2 out of 2)

Case Type	Passed	Total
Violation	1	1
Boolean	1	1
All Cases	2	2

State	Contract Violation	Test Name
Test1	ACCOUNT_TEST	
PASSED	NONE	test_withdraw
PASSED	NONE	*test_negative_amount

Fig. 5: Boolean and Violation Test Cases

1. The boolean valued routine `test_withdraw` in Fig. 5 is a boolean test case. The condition to be checked (`a.balance = 20`) is on the right side of an assignment to the `Result` entity (Eiffel's predefined return variable).

2. The routine `test_negative_amount` – a violation test case – has an expected contract violation when a negative amount is withdrawn from the account – this is an expected violation of the precondition. If there was a typo in the precondition (e.g. $amount \geq 0 \vee amount \leq balance$) then the test would fail, and we would be motivated to either change the test or the contracts (one of them must be wrong).

The relationship between the tests in Fig. 5 and the E-Tester framework is illustrated in Fig. 6.

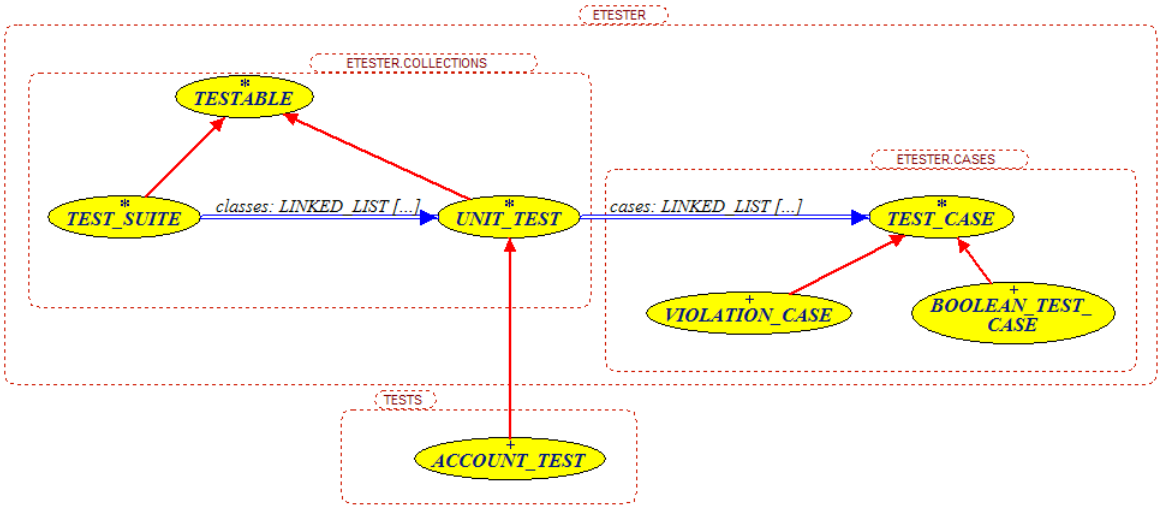


Fig. 6: Unit tests in `ACCOUNT_TEST` in context with E-Tester

Running the tests of Fig. 5 can be done using either E-Tester's command-line interface, or through the E-Tester GUI. The test results are reported to a file declared in the `make` creation feature (`tests.htm`) and a GUI interface runs the tests and displays the results (see bottom of Fig. 5). If all tests pass, a green bar is shown with test statistics. Boolean test successes and violation succeses are reported separately. As well, each test passed is listed. If even one test fails, a red bar is displayed and the failing tests are indicated. Any contract violations can be traced directly by running the debugger.

This scheme has the benefit of making expected contract violations simple to test. As well, there is no need for the multiplicity of `assert` statements (e.g., assertTrue) used in JUnit. The condition to be tested for is written directly as a predicate (on the right hand side of an assignment to `Result`) in the same way that one writes regular contracts.

A concrete set of test cases are gathered together in an effective class that inherits from deferred class `UNIT_TEST` (e.g., see class `ACCOUNT_TEST` in Fig. 5). As in JUnit,

each test case (e.g. `test_withdraw`) must be converted into an object to be stored in a list of test cases, i.e.,

`cases:LIST[TEST_CASE]`

as shown in Fig. 4. In contrast to JUnit, we cannot directly use the *Command* pattern because each test case has a different name, and we want the concrete class to contain all the cases. In E-Tester, we use Eiffel's agent mechanism instead.

The agent mechanism allows us to create an object that represents a routine such as the boolean case `test_withdraw`. This object is stored in the list `cases`, and the stored routine can executed at a later date when we are ready to execute all the tests. Each concrete test in `ACCOUNT_TEST` is added either as boolean case or as a violation case in the `make` routine (Fig. 5). Boolean cases are function routines (returning either true or false depending on the evaluation of `Result`). Violation cases are procedural routines such as `test_negative_amount` in Fig. 5.

To distinguish between boolean and violation tests, the BON class diagrams in Figs. 4 indicate two corresponding classes `BOOLEAN_CASE and VIOLATION_CASE` that are both descendants of `TEST_CASE*`. Class `TEST_CASE*` has a deferred routine `run` which may be used to run the actual test stored. The implementation for a violation case is

```
run is
    local error: BOOLEAN
    do
        if not error then
            case.apply
            passed := false
        end
    rescue
        passed := true
        error := true
        retry
    end
```

The deferred class `UNIT_TEST` has an attribute `cases` which is a `LIST` of either boolean or violation tests. A group of concrete test cases are written in a descendant of `UNIT_TEST`, in our case `ACCOUNT_TEST` (Fig. 5).

A test method such as the violation test case `test_negative_amount` is made into an object using the Eiffel's agent mechanism (to achieve something similar to the Command pattern in similar circumstances in JUnit). The routine `add_violation_case` is used to add a boolean test, and the routine `add_boolean_case` is used for boolean test cases (Fig. 4).

4.3 Using E-Tester in Test-Driven Development

E-Tester supports class-level testing and the expression of test suites. The framework provides classes that must be inherited by tests. The class *UNIT_TEST* represents unit tests, along with services for running all tests, outputting results to HTML, and adding and removing both boolean and violation tests. A *TEST_SUITE* class is also provided for representing sets of unit tests. The tests can either be run from the command line, or from the GUI provided with E-Tester; this GUI is being integrated with the Eiffel plug-in for Eclipse (see sequel).

When using E-Tester in a test-driven development process, there are three approaches that are usually taken.

1. *Typical test-driven.* The standard test-driven approach is followed, where unit tests are written, functionality is implemented to satisfy these tests, and periodically refactoring is carried out to simplify the code and improve understandability and extensibility. In this approach, the use of E-Tester is identical to test-driven approaches to using JUnit.

2. *Synergistic use of contracts and unit tests.* As discussed in [14], both contracts and unit tests are specifications of properties that need to be checked against working code. Sometimes it is preferable to write these specifications using contracts, and other times using tests, because of ease of expressiveness, criticality of the property being captured, and the availability of supporting tools to check the specifications. Clearly, tests provide a weaker form of specification (since a test will only capture a single scenario of use, whereas a contract will capture all scenarios). In this style of use of E-Tester, contracts and tests are *both* written to synergistically provide mutual feedback and relative consistency; that is, tests are written to check contracts (e.g., calling a routine with a *false* precondition), and contracts are written to improve confidence in tests. This approach hinges significantly on the use of violation test cases, and is a key difference between how we use E-Tester and we use JUnit.

3. *Collaborative tests.* Unit tests are normally used to validate and verify small parts of a design; JUnit and E-Tester excel at supporting this. Both these tools can also be used to write and execute tests that verify abstract behaviour, such as system-level tests. In particular, such tests could verify parts of the *collaborative* behaviour of objects in a system. These tests are critically important early in typical agile development processes, where units of functionality are determined in collaboration with a customer. Each unit is then implemented in a development increment. These units are typically described in terms of real-world entities and phenomena; in order to refine units, we typically use collaborations between objects[1]. Collaborative tests can be used to verify such abstract behaviour.

[1]This approach is also used in plan-driven processes, such as the Rational Unified Process, where use cases are refined by communication diagrams.

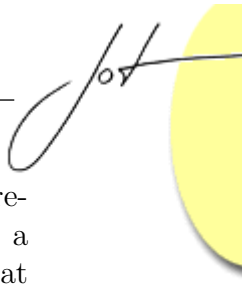


Fig. 7 contains an example of a collaborative test for a teller withdrawal request in a banking system. This test involves verifying the behaviour in a collaboration between an account and a teller transaction. If the code that is being tested implements pre- and postconditions (e.g., checking that a request for a withdrawal provides an acceptable amount that does not exceed the account balance), then the collaborative test can be used to exercise the contracts in the code. In this sense, the test and contract work synergistically, and running the test demonstrates that the code, contracts, and test are relatively consistent.

```
test_teller_withdrawal_request: BOOLEAN is
  local
    a: ACCOUNT; t: TELLER_TRANSACTION
  do
    -- setup, with initial balance 900
    create a.make("John Doe", 900);
    check a.balance = 900 end
    create t;

    -- test scenario
    t.request(a,500);
    t.withdrawal_request
    Result := a.balance = 400 and t.succeeded
  end
```

Fig. 7: Collaborative test

4.4 Comparison of E-Tester and JUnit

As demonstrated, the design of E-Tester differs from that of JUnit. Most of the design modifications are due to handling of contracts, but also because Eiffel supports limited forms of meta-referencing and reflection, unlike Java. Some of the key differences between E-Tester and JUnit are as follows. These differences suggest extensions that might be made to JUnit to better support unit testing involving contracts, e.g., as provided with iContract.

- When testing pre- and postconditions using JUnit, sequences of `assert` statements are typically used. In E-Tester, we use `Result := condition` to express tests[2], because it is simpler and more flexible – operations can thereafter be applied to the variable *Result* to express complex constraints on test results.
- Multiple tests can be supported in one test case in E-Tester. E-Tester's feedback will take developers directly to the line in the source code where the

[2]Recall that *Result* is an automatically declared variable in any function.

contract fails, because the Eiffel infrastructure supports such smart debugging of contracts. This would not be the case with JUnit unless substantial effort had been put into annotating code with assertions and documenting each assertion with informative messages.

- E-Tester makes use of violation cases because this eliminates the need to have `catch` clauses to deal with contract failures. This in turn simplifies the task of debugging and allows developers to more quickly highlight the contract that has failed. It also makes test cases generally simpler.
- JUnit uses reflection to extract test cases and add them to the test harness. Test cases are added manually with E-Tester since Eiffel does not support the flexible reflection mechanism of Java.
- E-Tester provides substantive support for adding comments to test cases; these comments are displayed when tests are run. With JUnit the only comment presented is the name of the test.
- JUnit is well integrated with Java IDEs, including Eclipse. Ongoing work is integrating E-Tester with Eiffel IDEs (see Section 6).

In terms of improving JUnit's support for contract testing, explicit support for violation test cases in the framework would be useful, as would reflective support for extracting contract labels from Java code in order to document which contracts have failed (and where they can be found).

5 COMPARISON WITH OTHER EIFFEL TESTING FRAMEWORKS

The main testing framework for Eiffel other than E-Tester is *getest*, due to Bezault [4]. *getest* is a command-line testing framework, and is compatible with a number of Eiffel compilers. Using *getest* requires developers to write individual test case classes that inherit from *TS_TEST_CASE*. This class will have argumentless procedures that exercise test scenarios, typically by making calls to a variety of inherited *assert* routines, e.g., *assert_equal*. For example, suppose that a developer wanted to test a class that provides a *string_concat* function. A test case would look like the following.

```
deferred class TEST_CONCAT1
inherit TS_TEST_CASE
feature
  test_concat is
  local c:CONCAT1
  do
    create c.make
```

```
      assert_equal("a+a", "aa", c.string_concat("a","a"))
      assert_equal("foo+bar", "foobar", c.concat("foo","bar"))
    end
  end
```

The first argument to *assert_equal* documents the test case, while the second provides expected output from a call. *getest* would be applied to the test case, and a table of results (tests passed, failed, aborted) would be presented in text format.

Key differences between *getest* and E-Tester are as follows.

- *getest* does not distinguish explicitly between violation and boolean test cases; developers may distinguish between the two via comments provided with the test case. The distinction between the two types of tests seems suitably complex that it is worthwhile to distinguish the two cases in the testing framework directly.
- Assertion failures in the program (i.e., pre- or postcondition failures) will result in test aborts unless explicit exception handlers are written.
- A configuration file must be written in order to run the test. This in effect allows simulation of test suites, but it is not as convenient as the wrapper provided with E-Tester since *getest* requires an external document to be written and managed.
- Results of testing are provided in text format rather than HTML or XML. This is a key point in terms of integration with additional tools.

There are other testing frameworks in Eiffel, e.g., EiffelUnit, which emphasises regression testing. EiffelUnit has been deprecated with the introduction of *getest*. Work at ETH Zurich is focusing on a Test Wizard [2] which will automatically generate test data from contracts. This should be seen as complementary to E-Tester and *getest* which serve to automate the testing process.

A related piece of work in the Java community is JMLAutoTest [20], which automatically generates tests from JML specifications. The results of applying JMLAutoTest can be used by JUnit to automate the testing process. JML's specification language is richer than that of Eiffel, and it is likely that the techniques used in JMLAutoTest can be applied to Eiffel; this will be directly relevant to the Test Wizard work mentioned above.

6 EDT: THE EIFFEL DESIGN TOOL FOR ECLIPSE

EDT is an under-development Eiffel plug-in for Eclipse; its development is being sponsored by IBM. The original design goals for EDT were to provide a full integrated development environment (with compilation, execution, and debugging facilities), as well as an Eiffel editor. This has recently changed based on discussions and

ongoing work on Test-Driven Development. In particular, the emphasis has changed to providing a lightweight IDE (still providing the requisite compile/run/edit facilities) that offers integrated testing support via E-Tester.

The current features of EDT include an Eiffel-aware editor, a content outliner, command completion, wizards for project construction, and partial integration with E-Tester. EDT currently supports an E-Tester view (which effectively presents the testing view of a project) with E-Tester producing its output in XML. Work remains on formatting the results of E-Tester so as to acquire the red/green bar deliverable discussed earlier.

Release information about EDT can be found at SourceForge [11].

7 CONCLUSIONS

We have presented the E-Tester unit testing framework for Eiffel, which supports testing of programs that make use of lightweight formal methods, i.e., contracts. The key observation that we made is that it is useful to distinguish violation from boolean test cases in such frameworks. This distinction may be useful in providing better unit testing support for other languages, e.g., JML, Java, etc. We have used E-Tester in numerous projects, e.g., implementing a metamodel directly in Eiffel [17].

We have mentioned ongoing work on integrating E-Tester with EDT, the Eclipse plug-in for Eiffel. Additional work is looking at test case derivation directly from Eiffel contracts, via the Test Wizard mentioned earlier, and using these test cases to improve the unit tests written by hand.

REFERENCES

[1] Ambler, S. *Test-driven Development*. www.agiledata.org/essays/tdd.html, accessed August 28, 2003.

[2] Arnout, K., X. Rousselot, and B. Meyer. Test Wizard: Automatic test case generation based on Design by Contract. se.inf.ethz.ch/people/arnout-/arnout, accessed May 2004.

[3] Beck, K. *Test-driven Development : by example*. Addison-Wesley, Boston, 2003.

[4] Bezault, E. *getest:* Gobo Eiffel Test. www.gobosoft.com/eiffel/gobo/getest/, accessed May 2004.

[5] Cohen, D., M. Lindvall, and P. Costa. Agile Software Development. Fraunhofer Center for Experimental Software Engineering, University of Maryland. 2003. http://citeseer.nj.nec.com/lindvall02empirical.html

[6] Fowler, M. and K. Beck. *Refactoring : improving the design of existing code*, Addison-Wesley, 1999.

[7] Gamma, E. and K. Beck. JUnit: A cook's tour. *Java Report*, p27-38, 1999.

[8] Kramer, R. iContract - the Java Design by Contract Tool. In *Proc. TOOLS 1998*, IEEE Press, 1998.

[9] Larman, C. and V. Basili. Iterative and Incremental Development: A Brief History. *Computer*, 36(6): p47-56, 2003.

[10] Leavens, G.T., K.R.M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion*, ACM, 2000. ftp://ftp.cs.iastate.edu/pub/techreports/TR00-15/.

[11] Makalsky, D. Eiffel Development Tool Plugin for Eclipse. http://sourceforge.net/projects/edt, accessed May 2004.

[12] Meyer, B. *Eiffel: the Language* (Second Edition), Prentice Hall, 1992.

[13] Meyer, B. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[14] Ostroff, J.S., D. Makalsky, and R.F. Paige. Agile Specification-Driven Design. In *Proc. Extreme Programming 2004*, LNCS, Springer-Verlag, 2004.

[15] Paige, R.F. and J.S. Ostroff. A Comparison of BON and UML. In *Proc. UML'99*, LNCS, Springer-Verlag, 1999.

[16] Paige, R.F and J.S. Ostroff. The Single Model Principle. *Journal of Object Oriented Technology*, 1(5): 2002.

[17] Paige, R.F, P.J. Brooke, and J.S. Ostroff. Test-Driven Development of a Reliable Executable Metamodel, submitted July 2004.

[18] Walden, K. and Nerson, J.-M. *Seamless Object-Oriented Software Architecture*, Prentice-Hall, 1995.

[19] Wayne, R. Don't let the Bugs Bite: Parasoft's Jtest and Jcontract. *Software Development*, p24-27, July 2003.

[20] Xu, G. and Z. Yang. JMLAutoTest: a Novel Automated Testing Framework based on JML and JUnit. In *Proc. FATES 2003*, LNCS 2931, 2004.

ABOUT THE AUTHORS

Jonathan S. Ostroff is an associate professor at York University, Toronto, Canada, where he leads research on object-oriented design, formal methods, and real-time software development. Email: jonathan@cs.yorku.ca

Richard F. Paige is a lecturer at the University of York, United Kingdom, where he works with the High-Integrity Systems Group and is a co-leader of the Software and Systems Modelling Team. He completed his PhD in Computer Science at the University of Toronto in 1997. paige@cs.york.ac.uk

David Makalsky is an M.Sc student at York University, Toronto, Canada, working on the Eiffel Design Tool (EDT), a plug-in for Eclipse. dm@cs.yorku.ca

Phillip Brooke is a senior lecturer at the University of Plymouth, United Kingdom, where he works with the Network Research Group. He completed his DPhil in Computer Science at the University of York in 1999. Email: philb@soc.plym.ac.uk