LIFT- The LIsp Framework for Testing

Gary King
Computer Science Department, LGRC
University of Massachusetts, Box 34610
Amherst, MA 01003-4610
gwking@cs.umass.edu
(413) 577-0669

Abstract

A sometimes overlooked value of software testing is that it enables fearless programming and constant refactoring. Since the design is often the first causuality of sitting down to code, having a complete test suite improves productively and allows for rapid change and experimentation. This report describes **LIFT**: the LIsp Framework for Testing. **LIFT** supports the construction of a hierarchical suite of tests that can be run in batch (for regression testing) or interactively (which suits the style of Lisp). The philosophy, design and use of **LIFT** are discussed and **LIFT** is compared with several other Lisp testing frameworks.

1 Introduction

A system with a complete test suite is a system that can be modified fearlessly. Because Lisp is so fluid and adept at rapid reshaping, it is more important than ever that Lisp projects have a complete test suite. However, Lisp's interactive nature encourages building tests piecemeal. While this allows for easy bottom up testing, it makes it hard to test the system as a whole. The following describes a method of building tests in Lisp that merges Lisp's interactive nature with complete regression testing. It is (very) loosely based on the Beck Testing Framework[Beck1999] and the JUnit Java testing framework developed by Kent Beck and Eric Gamma[Beck2000b]. The goal of all these frameworks is to provide a means of developing regression tests that:

1. can be built simply and quickly
2. encourages a code-a-little / test-a-little style of coding
3. can be run often

These tests help ensure that code is correct and, more importantly, allow developers the confidence to refactor and make changes because they know that they can verify the system easily.

The remainder of this document will first discuss the testing methodology in greater detail. Then we will outline the **LIFT** commands and their use and compare **LIFT** with other testing frameworks. Finally, we will cover some techniques that we have found to be helpful in our testing. One point to emphasize immediately is that **LIFT** is meant to add value to Lisp testing without adding (too much) work. Typically, a Lisp programmer will test her code by peppering it with:

```
#+Test
(assert (= 72 (factorial 6))
#+Test
(assert (= 1 (factorial 1))
```

In **LIFT**, this would look like:

```
(deftest test-factorial () ()
  (:tests
   ((ensure (= 720 (factorial 6))))
   ((ensure (= 1 (factorial 1))))))
```

Or, assuming that we've already defined the test suite:

```
(addtest (ensure (= 72 (factorial 6))))
(addtest (ensure (= 1 (factorial 1))))
```

Both `deftest` and `addtest` run any test-cases they define immediately so that the normal interactive Lisp style is upheld and supported. However, **LIFT** also provides the ability to run *all* of the test-cases non-interactively for complete unit and system testing.

2 Testing Methodology

It is no secret that testing is both important and neglected. The most common argument against testing is that it takes too much time and is too difficult. One answer to this is that, properly structured, testing is both easy and time saving. Furthermore, a system with a proper suite of tests can be radically refactored with confidence–the tests provide assurance that all remains well. For testing to work, it's important that tests be easy to build and that the coding of the system proceeds in parallel with the coding of the tests. This style makes it easier to write code because the code is easily verified. It also makes it easier to write tests because the writing of the code makes it clearer exactly what should be tested.

2.1 Testing Terminology

The Lisp Testing framework is built mainly around three macros (`deftest`, `addtest` and `ensure`) and a few methods used to run the tests. Before we describe these, however, it would be helpful to introduce some terminology:

test-case The smallest unit of testing. A test-case should prove one thing about a function, class, system or module.

fixture Each test-case operates within an environment provided by its fixture. The fixture is the code that prepares the environment for the test and that resets the environment after the test. Fixtures can be shared by many test-cases.

test-suite A test-suite is a hierarchical collection of test-cases and other test-suites. In **LIFT**, each test-suite in the hierarchy provides its own fixture.

error Tests can fail in two ways, a test error is the case when something completely unexpected occurs–for example, a system (or coding) error. These sorts of errors are tracked separately from test failures.

failure A test fails when the code being tested does not behave as expected. For example, the value returned by a function is not correct or a form that ought to produce a warning does not.

2.2 An Introduction to LIFT

In **LIFT**, test-suites are built around CLOS classes. The class provides a place to put slots necessary for the tests, methods to setup and tear-down the fixture

required for the tests in the suite and methods to run each test. The class hierarchy provides a natural way to group collections of test-cases and suites into larger collections of test-cases.

An example should make this clearer and give another view of **LIFT** syntax. The code below is a small portion of the tests developed for the Common Lisp Container Library (CLCL):

```
;;; ---------------------------------------------
;;; container tests
;;; ---------------------------------------------

(deftest test-containers)

;;; ---------------------------------------------
;;; binary-search-tree
;;; ---------------------------------------------

(deftest test-binary-search-tree (test-containers)
  ((b (make-container 'binary-search-tree)))
  (:setup (empty! b))
  (:tests
   (ensure (empty-p b))
   ((insert-item b 2)
    (ensure (not (empty-p b))))))

(addtest
 (insert-item b 2)
 (insert-item b 3)
 (delete-item b 2)
 (ensure-equal (size b) 1)))
```

The `deftest` macro creates a test-suite[1] and, optionally, the code required for test setup, test tear-down and the actual test-cases. `Addtest` adds a single new test-case to the most recently defined test-suite. The code above creates two test-suites (one for each `deftest`) and three test-cases (two from the `deftest` and one from the `addtest`). Since `test-containers` is a superclass of `test-`

[1] Actually, it creates a class which, for our purposes, is synonymous with the test-suite.

`binary-search-tree`, it "contains" all of its test-cases.[2] We can run the test-cases for binary-search-tree by evaluating:

```
(run-tests 'test-binary-search-tree)
```

We can run all of the container system's test-cases by evaluating `(run-tests 'test-containers)`. In either case, **LIFT** will report the total number of test-cases run and the number of failures and errors. If there are any problems, **LIFT** will provide a detailed report that makes it easy to see which tests need to be examined.

3 Macros, Functions and Variables

3.1 Macro addtest

Addtest is used to add another test to an existing test-suite. Its syntax is as follows:

```
(addtest
 [(test-class-name)]
 [name]
 [(:documentation <string>)]
 form*)
```

The test-class-name specifies the test-class to which the new test is added. If it is not specified, the new test will be added to the most recently defined test-class (i.e., the test-class created by the most recent evaluation of `deftest` or `addtest`). The `name`, `documentation` and `form` function as they do in `deftest`. For example:

```
;; This adds a new (unnamed) test-case to the
;; utilities test-suite.
(addtest (utilities)
 (ensure (equal (filter #'oddp '(1 2 3)) '(1 3))))

;; the ensure-warning form marks a test as failed if
;; a warning is NOT generated. This test-case is
;; named 'negative-factorials'
```

[2] `Test-containers` will also contain any test-cases from its other subclasses.

```
(addtest negative-factorials
 (ensure-warning (factorial -2)))

;; This is perhaps the simplest test form. It
;; creates an unnamed test. Ensure-equal compares
;; the results of the first form with the
;; (unevaluated) values of the rest of the
;; forms (see below for details)
(addtest (ensure-equal (factorial 0) 1))
```

3.2 Macro deftest

Deftest is an extended version of the defclass macro. As all of its parameters are optional, it does the best job it can parsing whatever it is given. The syntax is as follows:

```
(deftest test-suite-name
 [({supertest-name}*)]
 [({slot-specification}*)]
 [{test-clauses}*])
```

3.2.1 Supertest-names

Each supertest-name must have already been defined with deftest. The superclass `test-mixin` is added automatically to every test-class defined with deftest. `Test-mixin` provides the framework around which **LIFT** is built.

3.2.2 Slot specifications

Each slot-specifications consists of either a bare symbol or a list of the name of the slot, its initial value (or the keyword :unbound), and a symbol containing the letters I, R and A corresponding to :initarg, :reader and :accessor. Regular CLOS slot specification can also be used. The IAR symbol controls the generation of initarg, reader and accessor methods whose names will be the same as the slot name. For example:

```
(deftest foo (super-foo)
 ((slot-1 :unbound ir)
  (slot-2 34 a :type 'fixnum)
```

```
  slot-3))
```

Would expand into a class definition similar to:

```
(defclass foo (super-foo)
 ((slot-1
    :initarg :slot-1
    :reader slot-1)
  (slot-2
    :initform 34
    :accessor slot-2
    :type 'fixnum)
  slot-3))
```

3.2.3 Test clauses

The test-clauses can be one of:

:setup Used to specify code that will be run automatically before each test-case is executed. Setup code returns the test-environment to a known state. :Setup may only be specified once. Note that any initforms specified in the test-case slot definitions will automatically be made part of the test-case setup.[3]

:teardown Used to specify any code that should be run after each test-case is executed. For example, it might be prudent to close any files or free any resources. :Teardown may only be specified once.

:test The next form specifies a single test-case. :Test can be repeated as many times as desired.

:tests The next form is treated as a list of test-cases.

:documentation As in `defclass`, this clause stores documentation for the class being defined.

Each individual test-case has the following form:

```
([name] [(:documentation <string>)] form*)
```

[3] They are placed in a :before method.

The `name` is optional. If specified, it will be used to identify the test-case when reporting. The documentation string is also optional and is also used in reporting. When the test-case is run, the forms are evaluated as in `progn`. Typically, each test case will have one (or more) calls to the `ensure` macro (or one of its variants). Note that test slot-variables are available (as if with `with-slots`) within the body of all code generated with `deftest`. Here are some examples:

```
;; Setup a new testing superclass (for organization)
;; (note that we need specify neither superclasses
;; nor slots)
(deftest utilities)

;; this test class will be in the utilities test-suite,
;; is named 'required-slots' and has one slot named s.
;; We can use the slot in the test-case by naming
;; it (as in with-slots).
(deftest parse-slots (utilities)
 ((s (make-slot 'required)))
 (:test (required-slots (ensure (slot-required-p s)))))

;; This test class makes sure to close the
;; stream at the end of each test
(deftest file-tests (utilities)
 ((stream))
 (:setup (setf stream (open ...)))
 (:teardown (when (stream-open-p stream)
              (close stream))))
```

3.3 Macro ensure

Within a test-case, `ensure` is used to a assert that a given predicate holds true. If it does not, a test-failure will be logged for the test. All of the `ensure` macros can be used within the test environment or stand alone. In the former case, failures will be logged; in the latter, they will be reported interactively.

3.4 Macro ensure-error

You can use the `ensure-error` macro to verify that a particular form does indeed generate an error. Its syntax is:

```
(ensure-error &body body)
```

If the body does generate an error, the test succeeds. If it does not generate an error, then a test-failure will be logged. Here are some examples:

```
(addtest (examples)
 (ensure-error (warn "This test fails because a warning
is not an error.")))

(addtest (examples)
 (:documentation "This test will be logged as a
failure because no error will be generated.")
 (ensure-warning (= 2 2)))

(addtest (examples)
 (:documentation "This test succeeds!")
 (ensure-error (let ((x 0)) (print (/ 4 x)))))
```

3.5 Macro ensure-equal

The `ensure-equal` macro makes it convenient to ensure that a form returns the values you expect. Its syntax is:

```
(ensure-equal form value* &key (test #'equal))
```

It compares the (possibly multiple) value(s) returned by form to the values specified after the form using the test you specify. Here are some examples:

```
(addtest (utilities)
  (:documentation "Testing ensure-equal, should pass.")
  (ensure-equal (values "1" "2" "3") "1" "2" "3"
                :test #'string-equal))

(addtest (utilities)
  (:documentation "Testing ensure-equal, should fail")
  (ensure-equal (values "1" "2" "3") "1" "2" "3"
                :test #'eql))
```

3.6 Macro ensure-warning

You can use the `ensure-warning` macro to verify that a particular form does indeed generate a warning. Its syntax is:

```
(ensure-warning &body body)
```

If the body does generate a warning, the test succeeds. If it does not generate a warning, then a test-failure will be logged. Here are some examples:

```
(addtest (examples)
 (ensure-warning (warn "This test succeeds.")))

(addtest (examples)
 (:documentation "This test will be logged as a
failure because no warning will be generated.")
 (ensure-warning (= 2 2)))
```

3.7 Macro undeftest

This macro lets you removed a previously defined test. This can be helpful during interactive use if a test contains syntax errors or other problems. The syntax of `undeftest` is:

```
(undeftest [(test-class)] [test-name])
```

Both the `test-class` and the `test-name` parameters are optional. If they are unspecified, then the most recently defined test will be removed. If you specify only one parameter, then `undeftest` will assume that it is the test-name.

3.8 Function print-test-result

This function prints the results of testing. Its syntax is:

```
(print-test-result &optional stream test-result)
```

The stream parameter defaults to *standard-output*. The test-result parameter defaults to the most recently created test-result (from `run-tests`). If the `verbose?` option is used in `run-tests`, then `print-test-result` will be called automatically at the end of testing.

3.9 Function run-test

`Run-test` runs a particular test case.

3.10 Function run-tests

`Run-tests` runs all of the test-cases that have been defined for a test-class. By default, it will also run all of the test-cases for all sub-test-classes of a test-class. It has the following syntax:

```
(run-tests &key
 suite verbose? break-on-errors?
 do-subclasses? result)
```

The keywords are as follows:

suite This is the name (as a symbol) of the test-suite that you want to test. If you do not specity it, then it defaults to the most recently used (in `deftest` or `addtest`) test-class.

verbose? If `verbose?` is true, then the details of the test results will be printed after all of the test-cases have been run. It defaults to the value of *test-verbose?*.

break-on-errors? If this is true, then Lisp will go into the debugger if an error occurs while the test-cases are being run. If it is false, then errors will be logged and can be reported at the end of testing. `break-on-errors?` defaults to the values of *test-break-on-errors?*.

do-subclasses? When true, all of the test-cases in the subclasses of the test-suite will be run. This defaults to true.

result This should be a variable of type `test-result` into which the results of testing will be logged. If not specified, a new test-result will be created to hold the results.

3.11 Variable *test-break-on-errors?*

When *test-break-on-errors?* is true, the default behavior of `run-tests` will be to enter the debugger whenever an error is encountered during testing. This variable can be overriden by specifying a value for break-on-errors? when evaluating `run-tests`.

3.12 Variable *test-print-length*

test-print-length controls how test code is displayed when reporting errors and failures. It functions exactly like *print-length*.

3.13 Variable *test-print-level*

test-print-level controls how test code is displayed when reporting errors and failures. It functions exactly like *print-level*.

3.14 Variable *test-result*

This variable contains the summary information from the most recently evaluated `run-tests`. It is set by `run-tests` and used by `print-test-result`.

3.15 Variable *test-verbose?*

When *test-verbose?* is true, **LIFT** will be more informative about its activities during test creation and test evaluation. This variable can be overriden by specifying a value for verbose? when evaluating `run-tests`.

4 Comparisons

The most common Lisp testing technique is direct interaction with the Listener combined with a liberal sprinkling of `#+Test`'s in the code. Although this method has the benefit of immediate feedback, it lacks support for automated regression testing. Furthermore, since all testing is done in the same environment, the results of tests may come to depend on one another. Although using **LIFT** is not quite as simple as this direct style of testing (for example, you must specify what you expect the results to be instead of just looking at the output), it is almost as interactive, builds regression tests at the same time and provides each test case with a clean environment in which to run via fixtures.

The RT (Regression Test) package from MIT [RT1990] and the test framework available from Franz for Allegro Common Lisp [Franz2001] share many similarities. Both provide some support for interactive testing and the ability to combine tests for regression testing. Missing from both are **LIFT**'s ability to structure tests into a hierarchy of test-suites and, more importantly, the ability to factor out setup

and teardown routines into fixtures. This factoring allows common code to be reused and lets each test-case run in a clean environment.

5 Summary: Using LIFT

LIFT provides a framework for testing with enough flexibility that it can adapt to most environments. The following is a list of some guidelines that seem reasonable to us.

- If you have more than one test-suite (created by `deftest`, then specify the test-class-name in `addtest`. Since `addtest` creates a test for the most recently used test-class, you may find yourself adding tests into the wrong class otherwise.
- Write tests and code in parallel. The best time to write a test is probably before you write the code because that's when you are thinking most about what the code ought to do.
- Run your test suite often and correct failures as you find them. **LIFT** makes it easy to run all of your tests so that you can make sure that your most recent changes have not broken anything. This allows you to program with greater confidence.

LIFT provides a testing framework for Lisp that both supports traditional interactive testing and builds a suite of automated regression tests. **LIFT** is structured around a few simple commands making it easy to learn and easy to use.

6 Acknowledgments

Thanks go to David Westbrook and Brent Heeringa who provided valuable suggestions and commentary on **LIFT** and the **LIFT** documentation. This research is supported by DARPA contract DASG60-99-C-0074 and DARPA/AFOSR contract F49620-97-1-0485. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements either expressed or implied, of the DARPA or the U.S. Government.

References

[Beck1999] Kent Beck. Simple smalltalk testing: With patterns. 1999.

[Beck2000b] Kent Beck and Erich Gamma. Test infected:programmers love writing tests. 2000.

[Franz2001] Inc. Franz. A test harness for allegro cl. 2001.

[RT1990] Massachusetts Institute of Technology. Rt: Common lisp regression tester. 1990.