A CLOS Implementation of the JUnit Testing Framework Architecture: A Case Study

Sandro Pedrazzini
Canoo Engineering AG
sandro.pedrazzini@canoo.com

Abstract

There are different reasons why you would like to reimplement a tool or a framework using another language, trying to keep the same (maybe improved) functionality and the same design.
I did it for the JUnit testing framework, reimplemented in and for CLOS. In this article I will explain why, I will show how the tool has been integrated into the incremental and interactive Lisp way of developing and I will try to show a comparison between the two implementations.

1. Introduction

The tool presented in this article, "clos-unit", is a simple CLOS implementation of the famous JUnit framework, probably the most used testing framework within the Java developers' community.
So "yet another testing framework for lisp". Why then implementing clos-unit? There are many reasons for it, some very pragmatic, from a software development point of view, and others more didactical, from a computer science teaching point of view.
In the next chapters I will first explain the reasons that led me to the idea and necessity to implement clos-unit. In a second part I will quickly show how easy you can incrementally write, evaluate and add new test cases and suites with clos-unit. Then I will show the architecture and explain the main differences between the Java and the CLOS implementation, showing at the same time how easy and natural is the CLOS realization of some design patterns that generate complex code in Java.
During this article I use the terms CLOS and Lisp almost in an exchangeable way. The clos-unit tool has been implemented in CLOS, in order to follow the object-oriented architecture specified for JUnit, but, because Common Lisp Object System is embedded in Lisp, rather than representing its extension ([5],[7]), the term Lisp can also be used to represent it.

2. Reasons for implementing clos-unit

Here is a list of reasons for implementing a JUnit-based CLOS testing framework.

2.1. Regression tests

Regression tests is the concept of testing that asserts that everything that worked yesterday still works today.
Regression test is not always well integrated into the Lisp incremental natural way of coding, where the concept "code a little, test a little" is very effective, but old tests are not automatically collected and executed.
I needed an easy way to collect such already implemented tests and to execute them together delivering a clear result on correct running, failures and errors, still keeping the good habit of writing and evaluating single tests incrementally.
One could argue that regression test is not needed for functional programming, where each function can be tested fully independently during its realization. This is only partly true. Also in functional programming you can have some, very limited and localized ([5]) elements with side effects. Moreover regression tests allow to test overall consistency with data and the program at a higher abstraction layer, allowing refactoring at every layer.
Different test suites can be freely (and hierarchically) combined in order to build dependencies and test sequences used during different developing steps within a project.
The bottom-up incremental Lisp way of developing, combined with the JUnit test suite way of collecting tests, led to the idea to directly evaluate each test code while adding it to its test suite. This is important, because the framework builds and updates your regression test suite for you while you are evaluating your incremental tests as usual.

2.2. Simplicity

I wanted a framework that could help collecting tests without interfering with the business code, simple to use, with only minimal details to learn for the developer. The concept had to be coherently the same for simple tests as well as for more complex ones, i.e. tests needing fixtures or external resources.
Moreover, due to the Lisp (wonderful) overlapping between development time and run time, it must be possible to add new tests and update old tests without having to manage and delete old obsolete tests. In other words: each test can be automatically substituted by a new one.

2.3. Modularity

I knew the JUnit framework for having used it and for having analyzed its internal architecture. It has a core engine (model), managing and executing the tests, and a clear defined interface to the different presentations (views) of the model. Implementing clos-unit on JUnit design automatically guarantees that other people can implement different ways of presenting the results on the same core engine, as it may happen for JUnit.
This is a good way and reason to perform design reusability: modularity can be ported from one language to another one.

2.4. Using similar testing tools

This is only a minimal reason, but because I regularly switch between Java and CLOS programming, I wanted to fill the gap between the Java JUnit-based developing and the Lisp-based incremental developing, enabling a smaller context-switch and a consistent testing structure.
Organizing the tests in test cases and test suites in Java as well as in CLOS, having the opportunity to combine them in the same way, just helps to keep your testing code manageable and clear.

2.5. Developer's Motivation

Because using JUnit in Java is fun and programming in Lisp is fun, using clos-unit while programming in Lisp should mean double fun...
Adding the easy opportunity to collect tests for regression test gives the developer more confidence in refactoring his code.

2.6 Testing an existing design

The JUnit design has been conceived with a Java implementation in mind. But an architecture should be reusable independently from the programming language, at least to a certain degree of detail. I wanted to demonstrate it for a real case. So clos-unit is also meant as a software design case study. As I will show later, some patterns used in Java are not necessary or are much more easily implemented in a dynamic language as CLOS. Their use allows to keep a parallel architecture and therefore a similar internal documentation ([2]).

2.7. Comparing programming languages

The differences between Java and CLOS are quite clear, at least for CLOS developers. But it does not often happen to have the opportunity to reimplement exactly the same functionality with another programming language. Doing so you can measure the productivity of a language (one day implementation for the first clos version), the flexibility and, in this specific case, how dynamic a language is.
Taking each single design pattern used in the architecture we can analyze and compare the code step by step.

3. Using clos-unit

In this chapter I will first present the two new elements that must be known in order to use clos-unit: how to add a new test method to a test suite (with or without shared fixtures) and how to start one ore more test suites.

3.1. Test Methods

Each test is realized as method and must be part of a given class. The common class, shared among different test methods, determines the smaller

test suite and the context (initializations and finalizations) in which the test methods are run. More classes, with their test methods, can be combined, forming bigger suites, which sequences of methods can be run all together.

So here are the three steps that you need to perform in order to define a minimal test suite:

1. *Define your test class.*
   Your test class must be a subclass of "test-case".
   The simplest subclass does not need any particular instance variable, unless specifically needed for its test methods (I will show an example later).

2. *Add a new test method to your test class*.
   You add a new test method to your class using the macro "def-test-method". Adding a new test method also automatically performs the single test, in order to allow the usual Lisp incremental way of developing and test.

3. *Run the whole test suite*.
   As soon as you need to run all test methods together, call the method "textui-test-run".

3.2. Using Asserts

Each test method does not require any human judgement, in order to decide, whether the tests succeed or not.
When you want to check a value, you call one of the available "assert" functions, which all deliver true if the test succeeds.
For example, to test that the function "sum" actually delivers the right sum of two integers, you could use following assert expression in your test method:

```
(assert-true (eql (sum 1 2) 3))
```

There are other assert functions available. All functions have an optional parameter, used to show a message as output in case the test fails.
Here is the list of definitions:

```
(defun assert-true (expression &optional message)
   ...)

(defun assert-nil (expression &optional message)
   ...)

(defun assert-not-nil (expression &optional message)
   ...)

(defun assert-eql (arg1 arg2 &optional message)
   ...)

(defun assert-equal (arg1 arg2 &optional message)
```

```
...)
```

So here another way of expressing the check seen before for the sum function:

```
(assert-eql (sum 1 2) 3)
```

or, with output message:

```
(assert-eql (sum 1 2) 3 "Simple sum")
```

The clos-unit framework, similar to JUnit, distinguishes between failures and errors. The possibility of a failure is anticipated and checked for with assertions. Errors are unanticipated problems (errors in code), which do not have to do with the testing cases to check..

3.3. Simple Examples

If you look at the JUnit cookbook ([3]), you will find an extensive tutorial on how to use JUnit. The same tutorial can be used for clos-unit, easily adapting the examples in CLOS.
What I will show here are some shorter and stand alone examples. Their purpose being uniquely to show how to define a test suite and how to add and run new test methods.

We first define a test class, which will be used to collect all referring test methods. The simple version does not need any particular method or instance variable:

```
(defclass SAMPLE1-UNIT-TEST (test-case)
  ())
```

Then we can add new test methods. The methods must have only one parameter, which represents the dispatching object of the generic function, in this case the "sample1-unit-test" object.
The objects will use assert functions to be able to determine whether the test has a failure or not.

```
(def-test-method test-assert ((ob sample1-unit-test))
    (assert-true (eql (sum 1 2) 3)))

(def-test-method test-false ((ob sample1-unit-test))
    (assert-true (eql (sum 1 2) 4) "Test of failure"))
```

As soon as you evaluate the "def-test-method" expression, the new method is added to its test suite and the method body itself is run. In this way you have a immediate result, as well as adding "archiving" the method for regression tests.

3.4. Using fixtures

What if you have two or more tests that operate on the same or similar sets of objects? Or what if you have to open a common resource before each test method run and close it right after it?

Tests need to run against the background of a known set of objects or resources. This set is called a test fixture. When you are writing tests you will often find that you spend more time writing the code to set up the fixture than you do in actually testing values.
A big saving comes from sharing fixture code among test methods. Often, you will be able to use the same fixture for several different tests. Each case will send slightly different messages or parameters to the fixture and will check for different results.

When you have a common fixture, here is what you need to do:

1. Create a subclass of "test-case"
2. Add an instance variable for each part of the fixture
3. Override the "set-up" method to initialize the variables
4. Override "tear-down" method to release any permanent resources you allocated in "set-up"

Remember that set-up and tear-down are called each time before and after each test method call.

Here is an example on how to use the fixtures in clos-unit:

```
(defclass SAMPLE2-UNIT-TEST (test-case)
  ((db :initform nil :accessor db :initarg :db)
   (person :initform nil :accessor person :initarg :person)))

(defmethod set-up ((ob sample2-unit-test))
  (setf db (open-db-connection "my-test-db"))
  (setf person (make-new-person "john" "smith")))

(defmethod tear-down ((ob sample2-unit-test))
  (close-db-connection db)
  (setf db nil))
```

3.5. Combining test suites

Each test class corresponds to a suite, in other words when you add a new method to a test class you automatically add a new test to a test suite.
You can run an entire suite of tests calling one of the available xyx-test-run (currently only "textui-test-run") with the suite as parameter. Each element of the suite is a "test-case" object with a lambda expression corresponding to a test-method added previously.

But a test suite does not only contain "test-case" objects, it can contain any object inheriting from the "test" class, i.e. also "test-suite" objects. This allows to build free combinations of test suites hierarchically related.

3.6. User Interface

Right now there is just one way to start a suite of test methods together: calling the method "textui-test-run" with the corresponding suite as parameter. You can get access to a test class suite passing the class symbol name to the "get-suite" macro:

```
(textui-test-run (get-suite sample1-unit-test))
```

Here is the output you would get when the 2 test methods in 3.2 (test-assert and test-false) were added:

```
..F

FAILURES!!!
Run: 2   Failures: 1   Errors: 0

There was 1 failure:
1) TEST-ASSERT-FALSE: Test of failure
```

The output is similar as the output delivered by the JUnit text version. No graphical user interface has been implemented now, but the model-view internal architecture does not exclude it, as demonstrated in JUnit.

You can freely combine different test suites adding one to another using "add-test" and then starting the main one, in order to execute the whole list of tests:

```
(let ((composite-suite (make-instance 'test-suite)))
  (add-test composite-suite (get-suite sample2-unit-test))
  (add-test composite-suite (get-suite sample1-unit-test))

  (textui-test-run composite-suite))
```

Notice that no global variable or external reference is needed to manage the different test suites and their relations. A hidden class variable is responsible for this, but the user does not need to know it. This is another added value of this implementation.

4. Comparing the Implementations

In this last part I will consider the design of the framework and I will compare the two implementations (JUnit and clos-unit) for at least the main patterns.

Here are the discussed patterns:

- Template Method, used to ensure a skeleton in which each test method can automatically start common fixtures;

- Collecting Parameters, used to collect the results of each single test and show them at the end of the suite execution;

- Pluggable Selector, used in Java through the reflection mechanism to automate the insertion of each single test method as object into the suite and then call it as a command through an adapter.
  In dynamic languages like CLOS this is a straightforward operation.

- Composite: Used to freely combine different levels of suites and single test cases in a transparent way.

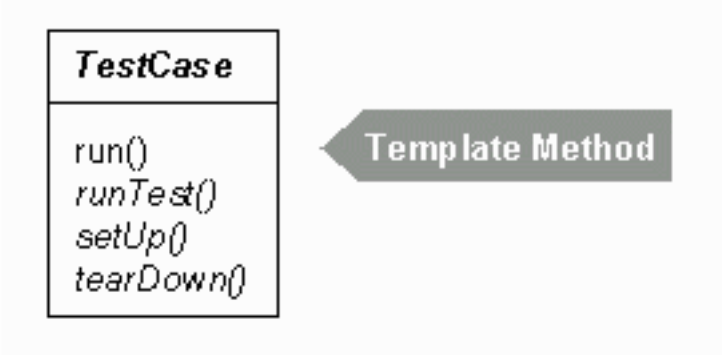
For a more general discussion on patterns for dynamic languages you can refer to [6].

4.1. Template Method

In JUnit the template method pattern has been used in order to give the developer a convenient place to put the fixture code and the test code ([3]).
The Template Method defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. The Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure ([4]).

There is in fact a common structure to all tests: they set up a test fixture, run some code against the fixture (test methods), check some results, and then clean up the fixture. This means that each test will run with a fresh fixture and the results of one test can't influence the result of another.
Quoting from the JUnit cook's tour document ([2]): "We want the developer to be able to separately consider how to write the fixture (set up and tear down) code and how to write the testing code. The execution of this sequence, however, will remain the same for all tests, no matter how the fixture code is written or how the testing code is written".

So here is the Template Method as described in JUnit cook's tour, represented within its abstract Java class *TestCase*:



To define a structure of an algorithm, in order to be able to customize it through subclassing is a feature also needed in languages like CLOS. To some extent this can be simplified with :before, :after and :around methods, without need to subclasses, but when the structure is more complex, defining "hot spots" (as in Template Method) to be redefined in subclasses is a valuable technique. Moreover, some needed bookkeeping are better managed through the "original" implementation of the pattern.
So here is the CLOS implementation of the Template Method pattern:

```
(defmethod run ((ob test-case))
  (set-up ob)
  (unwind-protect
    (run-test ob)
    (tear-down ob)))
```

And here are the empty fixtures as defined in test-case class:

```
(defmethod set-up ((ob test-case))
  ())

(defmethod tear-down ((ob test-case))
  ())
```

If you need the same fixtures for all test methods in your class, you just need to overwrite one or both fixture methods in your test class, as explained in paragraph 3.4.

4.2. Collecting Parameters

This pattern, coming from [1], has been used in JUnit in order to collect the results of each single test method.
The Collecting Parameter pattern suggests that when you need to collect results over several methods, you should add a parameter to the method and pass an object that will collect the results for you. That is why a new object has been created, TestResult, to collect the results of running tests.

Considering to follow the JUnit implementation, here is how we should implement the pattern in CLOS:

```
(defmethod run-on-test-result ((ob test-case) (res test-results))
  (start-test res ob)
  (run-protected res ob)
  (end-test res ob))
```

The surrounding methods "start-test" and "end-test" will pass information further to the right user interface listeners, in order to update their status after each test.
The method "run-on-test-result" should be called from the starting method "xyx-test-run " (in our case "textui-test-run"), which should create the test-results object and use it to show the final results. Here is an implementation of "textui-test-run":

```
(defmethod textui-test-run ((ob test))
  (let ((test-runner (make-instance 'textui-test-runner))
        (result (make-instance 'test-results)))
    (add-listener result test-runner)
    (run-on-test-result ob result)
    (print-results test-runner result)))
```

This is all right but following too much the imperative programming paradigm.
In Lisp we are more used to the functional programming paradigm, which is much cleaner and tends to eliminate every kind of side-effect. Lisp users are fortunate in being able to write code in both ways, whereas some languages are only suited to imperative programming.
In [5] a trick for transforming imperative programs into functional ones is explained. The trick is to realize that an imperative program is a functional program turned inside-out: "To find the functional program implicit in our imperative one, we just turn it outside-in".
Let's transform the method "run-on-test-result" from imperative to functional following this technique (of course also some other elements in the rest of the program must be consequently adapted):

```
(defmethod run-on-test-result ((ob test-case) (res test-results))
  (end-test
   (run-protected
    (start-test res ob) ob) ob))
```

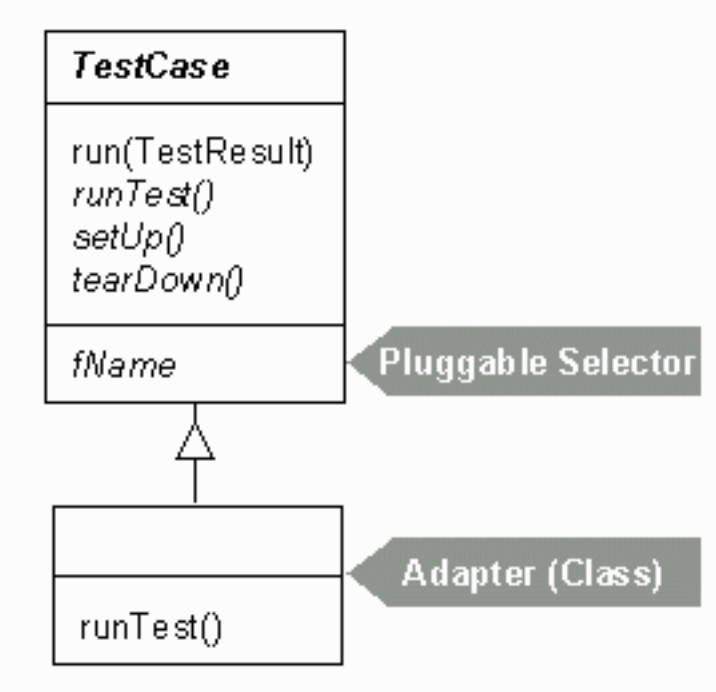
In other words, the Collecting Parameter pattern turns out to be a very implicit pattern (or non pattern) in functional programming.

4.3. Pluggable Selector

This will also turn out to be a very easy implementation in Lisp, whereas it needs some tricks in Java.
Let's first explain what is meant as pluggable selector ([1]).
In Java you need an interface to generically run the test methods. However, all test cases are implemented as different methods in the same class. This avoids the unnecessary proliferation of classes. A given test case class may implement many different methods, each defining a single test case. Each test case has a descriptive name. The test cases don’t conform to a simple command interface, which would be useful in order to be able to run all tests the same way. Different instances of the same Command class need to be invoked with different methods. Therefore the problem is to make all the test cases look the same from the point of view of the invoker of the test.
A possible solution could be the use of a single class which can be parameterized to perform different logic. The parameter is used to select a single test method and run it within a common interface (method "runTest in JUnit and "run-test" in CLOS).

In Java there is no notion of method selector. So the solution is to use the Java reflection API in order to invoke a method from a string representing the method’s name. During run-time for each test method of a given class a

new object of the same class is instantiated embedding the method name in form of string, able to execute the method behind a common interface (turned to a Command pattern through an Adapter). Here is the representation taken from JUnit cook's tour document:



This behaviour is much more easy and straightforward in Lisp: it is common to assign a lambda expression as a value to a variable. So each test method can be easily added to its suite without special tricks. We still embed it into an object (Command pattern) mainly for two reasons: first, we need to keep its name, for documentation in case of failure, second, to follow the initial idea to keep the architecture as near as possible to the JUnit one, for the reasons explained before.

For this operation I defined the transformation macro "def-test-method", more an extension of the language than a function of the program, able to add a defined test method to its corresponding suite, after embedding it into an instance of its "test-case" class.

```
(defmacro def-test-method
                (method-name class-name &body method-body)
  `(let ((,(caar class-name)
          (make-instance ',(cadar class-name)
            :name ',method-name)))
     (setf (method-body ,(caar class-name))
           #'(lambda() ,@method-body))
     (add-test (suite ,(caar class-name)) ,(caar class-name))
     (textui-test-run ,(caar class-name))))
```

The last line of code is used to execute the method body as soon as the new method is added, as we are used in Lisp incremental development.

4.4. Composite Pattern

Like the Template Method, also the Composite pattern can be useful in Lisp implementations, even though Lisp has more powerful and easier way to realize it than other languages: a list of lists, without using strong typing features, represents effectively the essence of the composite pattern (with some abstraction functions which allow a transparent use).

A test suite is in general a collection of tests. You can combine different test suites, so the element of a single suite can be single test methods, but can also be other test suites. We want to support suites of suites of suites of tests. In other words this represents a hierarchical data structure. The Composite pattern is used to combine and transparently use test suites. In clos-unit I implemented the Composite pattern following the definition given in [4].
The invoker of the tests doesn't have to care about whether it runs one or many test cases. Composite lets clients treat individual objects and compositions of objects uniformly.

The pattern tells us to introduce an abstract class which defines the common interface for single and composite objects. The class in clos-unit is "test", which does not need to represent the interface, but is needed in order to define a common class between "test-case" and "test-suite".
The rest of the pattern is then realized with the two subclasses "test-case" and "test-suite" and their corresponding methods "run-on-test-result", shown here:

```
(defmethod run-on-test-result ((ob test-case) (res test-results))
  (end-test
   (run-protected
    (start-test res ob) ob) ob))

(defmethod run-on-test-result ((ob test-suite) (res test-results))
  (mapc #'(lambda (composite)  ;;test-case or suite
            (run-on-test-result composite res))
        (tests ob)))
```

Each test-suite manages in its slot "tests" a list of "test" elements, i.e. "test-case" elements or "test-suite" elements. The second method shows the recursive character of the structure. If the element of the list is a test-case element, then the test-case version of "run-on-test-result" is called, otherwise the same test-suite "run-on-test-result" method is recursively called.

Conclusions

I have presented the implementation of clos-unit, a testing framework for CLOS programs. I tried to motivate the need of such implementation, following the JUnit architecture, i.e. the design of a very used Java based testing framework; I explained how to use it and I discussed some design and implementation issues, comparing them with the Java implementation.

The result is a tool which is simple to use during software development and easy to understand. The framework has been used and tested by different developers of the MCL community who gave comments and suggestions. It will be used as official tool in further works for [8] and other related projects.
From a didactical point of view (this is, after all, a case study) it has shown how portable a design can be and it has introduced some discussion issues on implementing pattern in different languages.
From a Lisp programming point of view it has once more demonstrated the productivity of the language. One transformation (macro def-test-method) was required to allow an easy use of the tool, the rest is a quick and easy implementation of the JUnit architecture.

References

[1] Beck Kent: Smalltalk Best Practice Patterns, Prentice Hall, 1997.

[2] Beck K. and Gamma E.: JUnit A Cook's Tour, in JUnit documentation, available at http://junit.sourceforge.net/doc/cookstour/cookstour.htm

[3] Beck K. and Gamma E.: JUnit Cookbook, in JUnit documentation, available at http://junit.sourceforge.net/doc/cookbook/cookbook.htm

[4] Gamma et al.: Design Patterns, Elements of Reusable Software Design, Addison Wesley, 1995.

[5] Graham Paul, On Lisp, Advanced Techniques for Common Lisp, Prentice Hall, 1994.

[6] Norvig Peter, Design Patterns in Dynamic Programming, available at http://norvig.com/design-patterns/, 1996.

[7] Steele Guy L. Jr.: Common Lisp, The Language, Second Edition, Digital Press 1990.

[8] Domenig M. and ten Hacken P: Word Manager, Olms Verlag, 1992