NST: A Unit Test Framework for Common Lisp

John Maraist

Smart Information Flow Technologies (SIFT, LLC)

TC-lispers, June 9, 2009

Outline

What is unit testing?

From wikipedia:

> *Unit testing is a software verification and validation method where the programmer gains confidence that individual units of source code are fit for use.*

- “Unit” refers to the basic elements of program design — procedures, functions, classes, etc.
- Unit tests should be independant of each other.
- Typically written by the programmer.
- Also usable as regression tests.

Basic concepts

Fixtures

- Named data values to which we apply tests.

Groups

- Collections of tests.

Tests

- One application of a criterion to values, usually to fixtures.

Criteria

- Named process of verification.

A simple NST example

Define two fixtures.

```
(def-fixtures simple-fixture
    (:documentation "Define two bindings")
  (magic-number 120)
  (magic-symbol 'asdfg))
(def-test-group simple-test-group
    (simple-fixtures)
  (def-test has-num
      (:eql magic-number)
    (factorial 5))
  (def-test has-sym
      (:eq magic-symbol)
    'asdfh))
```

A simple NST example

Define a group of tests.

```
(def-fixtures simple-fixture
    (:documentation "Define two bindings")
  (magic-number 120)
  (magic-symbol 'asdfg))
(def-test-group simple-test-group
    (simple-fixtures)
  (def-test has-num
      (:eql magic-number)
    (factorial 5))
  (def-test has-sym
      (:eq magic-symbol)
    'asdfh))
```

A simple NST example

The groups' tests are given the two fixtures.

```
(def-fixtures simple-fixture
    (:documentation "Define two bindings")
  (magic-number 120)
  (magic-symbol 'asdfg))
(def-test-group simple-test-group
    (simple-fixtures)
  (def-test has-num
      (:eql magic-number)
    (factorial 5))
  (def-test has-sym
      (:eq magic-symbol)
    'asdfh))
```

A simple NST example

Test names.

```
(def-fixtures simple-fixture
    (:documentation "Define two bindings")
  (magic-number 120)
  (magic-symbol 'asdfg))
(def-test-group simple-test-group
    (simple-fixtures)
  (def-test has-num
      (:eql magic-number)
    (factorial 5))
  (def-test has-sym
      (:eq magic-symbol)
    'asdfh))
```

A simple NST example

Test criteria.

```
(def-fixtures simple-fixture
    (:documentation "Define two bindings")
  (magic-number 120)
  (magic-symbol 'asdfg))
(def-test-group simple-test-group
    (simple-fixtures)
  (def-test has-num
      (:eql magic-number)
    (factorial 5))
  (def-test has-sym
      (:eq magic-symbol)
    'asdfh))
```

A simple NST example

Forms to be tested.

```
(def-fixtures simple-fixture
    (:documentation "Define two bindings")
  (magic-number 120)
  (magic-symbol 'asdfg))
(def-test-group simple-test-group
    (simple-fixtures)
  (def-test has-num
      (:eql magic-number)
    (factorial 5))
  (def-test has-sym
      (:eq magic-symbol)
    'asdfh))
```

A simple NST example

This test passes.

```
(def-fixtures simple-fixture
    (:documentation "Define two bindings")
  (magic-number 120)
  (magic-symbol 'asdfg))
(def-test-group simple-test-group
    (simple-fixtures)
  (def-test has-num
      (:eql magic-number)
    (factorial 5))
  (def-test has-sym
      (:eq magic-symbol)
    'asdfh))
```

A simple NST example

This test fails.

```
(def-fixtures simple-fixture
    (:documentation "Define two bindings")
  (magic-number 120)
  (magic-symbol 'asdfg))
(def-test-group simple-test-group
    (simple-fixtures)
  (def-test has-num
      (:eql magic-number)
    (factorial 5))
  (def-test has-sym
      (:eq magic-symbol)
    'asdfh))
```

Fixtures

```
(def-fixtures FIXTURE-NAME
    ( [ :uses USES ]
      [ :assumes ASSUMES ]
      [ :inner INNER ]
      [ :documentation DOCUMENTATION ] )
  (NAME FORM)
  (NAME FORM)
   ...
  (NAME FORM))
```

Fixtures and free variables

Keyword argument `:assumes`

- Names the variables occurring free in the fixture bodies.

```
(def-fixtures derived-fixtures-1
    (:assumes (magic-number))
  (magic-bonus (* 6 magic-number)))
```

Keyword argument `:uses`

- Names the other fixture sets whose bound names occur free in these fixtures bodies.

```
(def-fixtures derived-fixtures-2
    (:uses (simple-fixtures))
  (magic-bonus (* 6 magic-number)))
```

Fixtures and declarations

Keyword argument `:inner`

- Provides additional declarations for fixture bodies.

```
(def-fixtures internal-magic
    (:inner ((special magic-internal-state)))
  (state-head (car magic-internal-state))
  (state-snd  (cadr magic-internal-state)))
```

- Should be renamed `:declare`

Keyword argument `:outer`

- Ignored — useless hangover from an earlier design.

Test groups

```
(def-test-group NAME (FIXTURE ··· FIXTURE)
  [ (:setup FORM FORM ··· FORM) ]
  [ (:cleanup FORM FORM ··· FORM) ]
  [ (:each-setup FORM FORM ··· FORM) ]
  [ (:each-cleanup FORM FORM ··· FORM) ]
  TEST
  TEST
  ...
  TEST)
```

Setup and cleanup forms

```
(def-test-group show-setup ()
  (:setup   (write " S group"))
  (:cleanup (write " C group"))
  (:each-setup
    (write "  S-each group"))
  (:each-cleanup
    (write "  C-each group"))
  (def-check ts1 :pass
    (write "   ts1"))
  (def-check ts2 :pass
    (write "   ts2")))
```

```
Running group show-setup
 S group
  S-each group
   ts1
  C-each group
  S-each group
   ts2
  C-each group
 C group
Group show-setup:
        2 of 2 passed
```

Tests and groups

```
(def-test (NAME [ :setup FORM ]
                [ :cleanup FORM ]
                [ :fixtures (FXTR ... FXTR) ] )
    criterion
  FORM)

(def-test NAME
    criterion
  FORM)
```

Fixtures for individual tests

```
(def-fixtures simple-fixture ()
  (magic-number 120)
  (magic-symbol 'asdfg))

(def-test-group some-magic ()
  (def-test no-magic :true
    (not (boundp 'magic-number)))

  (def-test (with-magic
                :fixtures (simple-fixture))
      (:eql 120)
    magic-number))
```

Setup, cleanup on individual tests

```
(defparameter for-setup 0)

(def-test-group setup-cleanup ()
  (:setup (setf for-setup 1))
  (:cleanup (setf for-setup 0))
  (def-test a-sc-for-setup-1 (:eql 1) for-setup)
  (def-test (sc-for-setup-2
             :setup (setf for-setup 2)
             :cleanup (setf for-setup 1))
      (:eql 2)
    for-setup)
  (def-test z-sc-for-setup-1 (:eql 1) for-setup))
```

Setup, cleanup on individual tests

Recently noticed bug: `each-setup` for the group and `setup` for the test are not applied in the order we'd hope.

```
(def-test-group each-setup-cleanup ()
  (:each-setup (setf for-setup 2))
  (:each-cleanup (setf for-setup 0))
  (def-test (sc-for-setup-2
               :setup (setf for-setup 3)
               :cleanup (setf for-setup 2))
      (:info "This is a known bug" (:eql 3))
    for-setup))
```

Equality criteria

```
(def-test eql1 (:eql 2)
  (cadr '(1 2 3)))
```

```
(def-test eq1 (:eq 'a)
  (car '(a 3 c)))
```

And similarly for `equal` and `equalp`.

```
(def-test sym1 (:symbol a)
  (car '(a b c)))
```

Predicates and transformations

Boolean-valued functions can be used as test criteria:

```
(def-test pred1 (:predicate numberp) 3)
```

Forms can be altered before testing:

```
(def-test applycheck
    (:apply cadr (:eql 10))
  '(0 10 20))
```

List criteria

Expect every element of a list to pass a criterion:

```
(def-test each1 (:each (:predicate evenp))
  '(2 4 8 20 100))
```

Apply different criteria to respective elements:

```
(def-check seqcheck
    (:seq (:predicate symbolp)
          (:eql 1)
          (:symbol d))
  '(a 1 d))
```

Information and warnings

Additional information for results:

```
(def-test known-bug
    (:info "Known bug" (:eql 3))
  4)
```

Passing with a warning:

```
(def-test known-bug
    (:warn "~d is not a perfect square" 5)
  5)
```

Compound criteria

Negating another criterion:

```
(def-test not1 (:not (:symbol b)) 'a)
```

Passing all of a set of criteria:

```
(def-check not1 ()
    (:all (:predicate even-p)
          (:predicate prime-p))
  2)
```

And similarly, `:any` for at least one of a set of criteria.

Let’s argue about multiple values!

In Lisp:

- Functions can return multiple values.
- But paying attention to the “extras” is optional.
- Accessing “extras” takes a little extra effort.

In NST:

- Even if they’re ingorable, extra values must be *correctly implemented*.
- So when validating a function, NST expects all values to be accessed (and presumably validated).
- Ignoring “extras” takes a little extra effort.
- A mismatch between the received and expected values is an error.

Let's argue about multiple values!

In Lisp:

- Functions can return multiple values.
- But paying attention to the "extras" is optional.
- Accessing "extras" takes a little extra effort.

In NST:

- Even if they're ingorable, extra values must be *correctly implemented*.
- So when validating a function, NST expects all values to be accessed (and presumably validated).
- Ignoring "extras" takes a little extra effort.
- A mismatch between the received and expected values is an error.

Criteria for multiple values

Applying different criteria to respective values:

```
(def-test values1 (:values (:symbol a) (:eq 'b))
  (values 'a 'b))
```

Treating values as a list:

```
(def-test value-list1
    (:value-list (:seq (:symbol a) (:eq 'b)))
  (values 'a 'b))
```

Dropping extra values:

```
(def-test no-values1 (:drop-values (:symbol a))
  (values 'a 'b 'c))
```

Multiple-values and multiple-argument predicates

Multiple values are handled as additional arguments to the functions underlying criteria:

```
(def-test tricky-1 :eql
  (round 5 4))

(def-test values-drop1
    (:apply (lambda (x y)
               (declare (ignorable y))
              x)
      (:symbol a))
  (values 'a 'b))
```

Other criteria

For a full list of criteria, see the manual.

- Additional basic checks.
- `:err` — expecting an error.
- `:permute` — list permutation.
- Simple checks for vectors, slots.
- `:perf` — timing form evaluation.

Defining new criteria

NST provides three macro-style mechanisms for defining a new criterion:

- By describing how it translates to another criterion.
- By describing how it maps values to a results expression.
- By describing how it maps a form (which would evaluate to a list of values) to a results expression.

In fact, all of the built-in criteria use one of these three mechanisms.

Defining new criteria

- By translation:

```
(def-criterion-alias (:symbol name)
  `(:eq ',name))
(def-criterion-alias (:drop-values criterion)
  `(:apply (lambda (x &rest others)
             (declare (ignorable others))
             x)
     ,criterion))
```

- By a map from values to a results expression: `:true`, `:eql`, `:predicate`, `:info`.
- By a map from a form to a results expression: `:not`, `:perf`, `:err`.

Running NST

- **From the REPL.** NST provides a command-line interface for running tests and inspecting results, with top-level aliases for some supported Lisps.

  ```
  :nst :help
  ```

- **From ASDF.** NST provides an ASDF system class, providing automatic `test-op` methods.
- **Output via JUnit.** NST can generate JUnit-compatible XML for GUI browsing of results.

Running NST

- **From the REPL.** NST provides a command-line interface for running tests and inspecting results, with top-level aliases for some supported Lisps.

  ```
  :nst :help
  ```

- **From ASDF.** NST provides an ASDF system class, providing automatic `test-op` methods.
- **Output via JUnit.** NST can generate JUnit-compatible XML for GUI browsing of results.

Running NST

- **From the REPL.** NST provides a command-line interface for running tests and inspecting results, with top-level aliases for some supported Lisps.

  ```
  :nst :help
  ```

- **From ASDF.** NST provides an ASDF system class, providing automatic `test-op` methods.
- **Output via JUnit.** NST can generate JUnit-compatible XML for GUI browsing of results.

A little bit of background

A number of Lisp language features greatly simplified writing NST.

- Macros.
- Compile-time code execution.
- Multiple inheritance.
- Sophisticated method dispatch.

Macros

The top-level NST forms all defined by macros:

- Components not evaluated, but rewritten into other forms which (may be) evaluated.
- The typical Lisp way of writing language extensions.
- Allows a direct specification of the test.

Compile-time code execution

Part of the macro-expansion of tests involves translating criteria into Lisp code.

- Criteria definitions expand to method definitions.
- So macro-expanding tests involves running Lisp at compile-time.
- Moreover, the required routines may be found in the same code file as the tests they help to expend.

Multiple inheritance

Lisp classes may have more than one superclass, and moreover may inherit method implementations from different superclass hierarchies.

- Unlike e.g. Java, where all but one superclass hierarchies may define only abstract methods.

NST uses multiple inheritance for fixture application.

- Fixtures, groups and tests all translate to classes.
- Fixtures make their bindings via methods of their class.
- Groups' and tests' classes are subclasses of the classes corresponding to the fixtures they apply.
- So they inherit the fixture-binding methods.

Dynamic variable scope

Lisp offers both *dynamic* and *static* scoping for local variables.

- Determine the rules as to how we know what value-binding a program variable should have.
- If this idea is new to you, then you're probably used to static scope rules.
- Simple dynamic scoping example:

```
(defun print-x ()
  (declare (special x))
  (write x))
(defun x-as-3 ()
  (let ((x 3))
    (declare (special x))
    (print-x)))
```

Method dispatch

Lisp has a very rich notion of method dispatch.

- Typically, dispatch chooses exactly one method for a particular object.
- In Lisp, there are many available *method combinations* to map a method invocation to some assemblage of the corresponding method definitions.
    - “Standard” principle method override.
    - “Around” methods wrap other methods.
    - “Before” and “after” methods.
    - Sequential execution of several methods, and various ways of combining their results.

Around-methods

Used with dynamic scoping rules for fixture definitions:

```
(def-fixtures simple-fixture ()
  (magic-number 120) (magic-symbol 'asdfg))
```

becomes something like:

```
(defmethod run-group :around ((gr simple-fixtures))
  (let ((magic-number 120) (magic-symbol 'asdfg))
    (declare (special magic-number magic-symbol))
    (call-next-method)))
```

Before- and after-methods

Used for `:setup` and `:cleanup` arguments:

```
(def-test (sc :setup (setf for-setup 2)
              :cleanup (setf for-setup 1))
     (:eql 2)
   for-setup)
```

becomes, in part, something like:

```
(defmethod run-test :before ((test sc))
  (setf for-setup 2))
(defmethod run-test :after ((test sc))
  (setf for-setup 1))
```

Fwrappers

Fwrappers are a Allegro Common Lisp feature.

- Like an around-method for non-generic functions.
- Useful for extending third-party code, etc.
- But for code we're writing ourselves, no clear advantage over around-methods.

Gensym abuse

In one version, NST solved potential name conflicts by naming almost all intermediate structures via `gensym`.

- Very effective at avoiding accidental name conflicts.
- Very secure against interference with testing internals.
- Very secure against debugging.
- Very secure against examination and understanding.
- Very secure against extension.

Gensym abuse

In one version, NST solved potential name conflicts by naming almost all intermediate structures via `gensym`.

- Very effective at avoiding accidental name conflicts.
- Very secure against interference with testing internals.
- Very secure against debugging.
- Very secure against examination and understanding.
- Very secure against extension.

Gensym abuse

In one version, NST solved potential name conflicts by naming almost all intermediate structures via `gensym`.

- Very effective at avoiding accidental name conflicts.
- Very secure against interference with testing internals.
- Very secure against debugging.
- Very secure against examination and understanding.
- Very secure against extension.

Static class data

NST must be able to retrieve information associated with fixtures, groups and tests.

- We compile fixtures, groups and tests to classes.
- We'd prefer this information to be available *statically*, with reference to the class but without necessarily referencing a particular object.

Static class data

With `:static` allocation for class slots, accessors still require an object instance.

```
(defclass zz () ((aa :allocation :class
                     :accessor zz-aa)))
(setf z1 (make-instance 'zz)
      z2 (make-instance 'zz))
(setf (zz-aa z1) 5)
(pprint (zz-aa z1))  ;; displays 5
(pprint (zz-aa z2))  ;; displays 5
(setf (zz-aa z2) 6)
(pprint (zz-aa z1))  ;; displays 6
```

We'd like a way to store & retrieve class data *without* an object instance.

Class metaobjects for static class data

One approach we tried was via the *metaclass*.

- Lisp models a program's structures via standardized Lisp objects.
- By default, the definition of a class is stored in an instance of `standard-class`.
- But we can extend `standard-class`, and include static data as fields in the metaclass.

```
(defclass mzz (standard-class)
  ((aaa :accessor zzz-aaa)))
(defclass zz () ()
  (:metaclass mzz))
(setf (zzz-aaa (find-class 'zz)) 5)
```

Prefer object prototypes

We later discovered the preferred style of using *class prototypes*.

```
(defclass zz () ((aa :allocation :class
                     :accessor zz-aa)))
(mop:finalize-inheritance (find-class 'zz))
(defun aa ()
  (zz-aa (class-prototype (find-class 'zz))))
(defun set-zz-aa (x)
  (setf (zz-aa (mop:class-prototype
                  (find-class 'zz)))
        x)
  x)
```

NST internal packages

Fixtures correspond to two classes:

- One with methods for applying fixtures to a group once.
- One with methods for applying fixtures individually to each test within a group.

Right now, we create a name for this class.

```
(def-fixtures pkg::gname |# ... |#
```

becomes (in part) something like

```
(defclass nst-fixture-test-class-names::pkg///gname
    () ())
```

Package abuse?

We would not need special packages for internal class names if the classes simply didn't have names.

- By using MOP function calls directly (as opposed to `defclass`) one can create *anonymous* classes.
- The internal classes themselves can be made available as static information of the reference class.

A MOP-like model

Not just macros, but also functions.

- À la `defclass` and `ensure-class`.
- Also suggests: multiple “front-ends” (macros) translating to the single backend.

Error reporting

Is there actually any programming tool where “improve error messages” is *not* on the to-do list?

- Errors arising from bad use of criteria (such as treating a list with a vector criterion) are not clearly marked as testing errors, rather than test failures.
- General difficulty in good error analysis.

QuickCheck

QuickCheck originated in Haskell as a framework for generating randomized tests of programmer-stated program invariants.

- `(equal (reverse (reverse x)) x)`
- `(if (< x y) (eql (max x y) y))`

Some version of QuickCheck might be useful in NST.

- Side-effects
- Cannot rely on types for direction.

Further integration

- Better control of NST options through ASDF.
- Automated XML/JUnit output.
- Other output formats.

If you want to really learn Lisp...

Writing a unit tester is an effective way to see many interesting areas of Lisp!

- Subtle macro authoring.
- Evaluation-time issues.
- Under the hood of the class system.

Is this actually unit testing?

Returning to wikipedia's definition of unit testing:

> *A unit is the smallest testable part of an application. In procedural programming a unit may be an individual program, function, procedure, etc., while in object-oriented programming, the smallest unit is a class, which may belong to a base/super class, abstract class or derived/child class.*

- Is our unit the expression?
- Is there a more natural way to structure tests around the unit of a function?
- Lisp is *also* object-oriented — should a Lisp unit test package offer a more natural framework for classes and methods?

Proof in the practicality

Future directions aside, we believe that NST is now, already, a useful tool.

- Has been used on several large projects already.
  - The SHOP2 planner.
  - A plan recognizer.
  - Language interpreter.
  - Security software.
- Reasonable mature and solid.
- Actively maintained.
- Open-sourced.