Eunit in Practice

Richard Carlsson
Klarna AB

Eunit / other xUnit frameworks

- Most xUnit frameworks rely on OOP
  - Test classes inherit from framework classes
  - Erlang does not have objects and inheritance
- Eunit is based around funs, Erlang data structures (lists, tuples) and macros
- EUnit tends to be much less verbose!

A normal project layout

```
.../myproject/
              Makefile

              src/
                  *.{erl,hrl}

              ebin/
                  *.beam
```

Build environment setup

```
# A simple Makefile
ERLC_FLAGS=
SOURCES=$(wildcard src/*.erl)
HEADERS=$(wildcard src/*.hrl)
OBJECTS=$(SOURCES:src/%.erl=ebin/%.beam)
all: $(OBJECTS) test
ebin/%.beam : src/%.erl $(HEADERS) Makefile
	erlc $(ERLC_FLAGS) -o ebin/ $<
clean:
	-rm $(OBJECTS)
test:
	erl -noshell -pa ebin \
	 -eval 'eunit:test("ebin",[verbose])' \
	 -s init stop
```

A project-global header file

```
%% File: src/global.hrl

-include_lib("eunit/include/eunit.hrl").
```

A minimal module

```
%% File: src/empty.erl

-module(empty).

-include("global.hrl").


%% this will be automatically detected as a test

a_test() -> ok.
```

Compile and run tests

```
$ make
erlc  -o ebin/ src/empty.erl
erl -noshell -pa ebin \
    -eval 'eunit:test("ebin",[verbose])' \
    -s init stop
====================== EUnit ======================
directory "ebin"
  empty:a_test (module 'empty')...ok
  [done in 0.007 s]
===================================================
  Test successful.
$
```

Auto-exported test functions

```
1> empty:module_info(exports).
[{a_test,0},{test,0},{module_info,0},
{module_info,1}]
2> empty:test().
  Test passed.
ok
3> eunit:test(empty).
  Test passed.
ok
4> empty:a_test().
ok
5> eunit:test({empty,a_test}).
  Test passed.
ok
6>
```

Disabling tests

- We can set ERLC_FLAGS=-DNOTEST in the makefile to compile without tests

```
# A simple Makefile
ERLC_FLAGS=-DNOTEST
...
```

- We could also set up a special “make release” target to build a version without tests, e.g.:

```
release: clean
    $(MAKE) ERLC_FLAGS="$(ERLC_FLAGS) -DNOTEST"
```

Recompile with tests disabled

```
$ make
erlc -DNOTEST -o ebin/ src/empty.erl
erl -noshell -pa ebin \
    -eval 'eunit:test("ebin",[verbose])' \
    -s init stop
====================== EUnit ======================
directory "ebin"
  module 'empty'
  [done in 0.004 s]
  There were no tests to run.
$
```

Test functions gone

```
1> empty:module_info(exports).
[{module_info,0},{module_info,1}]
2> eunit:test(empty).
  There were no tests to run.
ok
3>
```

Disabling tests by default

- We can define NOTEST in the global header file, to default to compilation without tests:

```
%% File: src/global.hrl

-define(NOTEST, true).
-include_lib("eunit/include/eunit.hrl").
```

- In this case, we need to define TEST in the makefile, to override NOTEST
- We could have a special makefile target that builds a test version.
- Pick a default that suits you

Test code independency

- Your test code is yours
  - Macros from eunit.hrl do not require license
  - EUnit parse transforms do only trivial things
- Compiled tests do not require EUnit to run
  - Can be run manually or from your own code
  - EUnit just makes running and reporting easier
- Possible to recompile without EUnit available
  - Trivial to write do-nothing replacement macros

Getting started with tests

```
%% File: src/fib.erl
-module(fib).
-export([f/1]).
-include("global.hrl").

f(0) -> 1;
f(1) -> 1;
f(N) when N > 1 -> f(N-1) * f(N-2).

f_test() ->
    1 = f(0),
    1 = f(1),
    2 = f(2).
```

Badmatch, but where?

```
====================== EUnit ======================
directory "ebin"
  fib: f_test (module 'fib')...*failed*
::error:{badmatch,1}
  in function fib:f_test/0

===================================================
  Failed: 1.  Skipped: 0.  Passed: 0.
```

Keep tests small and separate

```erlang
%% File: src/fib.erl
-module(fib).
-export([f/1]).
-include("global.hrl").

f(0) -> 1;
f(1) -> 1;
f(N) when N > 1 -> f(N-1) * f(N-2).

f0_test() -> 1 = f(0).

f1_test() -> 1 = f(1).

f2_test() -> 2 = f(2).
```

...to make the bugs easy to spot

```
====================== EUnit ======================
directory "ebin"
  module 'fib'
    fib: f0_test...ok
    fib: f1_test...ok
    fib: f2_test...*failed*
::error:{badmatch,1}
  in function fib:f2_test/0

    [done in 0.024 s]
===================================================
  Failed: 1.  Skipped: 0.  Passed: 0.
```

Asserts give more detail

```
%% File: src/fib.erl
-module(fib).
-export([f/1]).
-include("global.hrl").

f(0) -> 1;
f(1) -> 1;
f(N) when N > 1 -> f(N-1) * f(N-2).

f0_test() -> ?assertEqual(1, f(0)).

f1_test() -> ?assertEqual(1, f(1)).

f2_test() -> ?assertEqual(2, f(2)).
```

...which helps a lot

```
====================== EUnit ======================
directory "ebin"
  module 'fib'
    fib: f0_test...ok
    fib: f1_test...ok
    fib: f2_test...*failed*
::error:{assertEqual_failed,[{module,fib},
                           {line,14},
                           {expression,"f ( 2 )"},
                           {expected,2},
                           {value,1}]}
  in function fib:'-f2_test/0-fun-0-'/1
```

Multiple asserts in one function

```
%% File: src/fib.erl
-module(fib).
-export([f/1]).
-include("global.hrl").

f(0) -> 1;
f(1) -> 1;
f(N) when N > 1 -> f(N-1) * f(N-2).

f_test() ->
    ?assertEqual(1, f(0)),
    ?assertEqual(1, f(1)),
    ?assertEqual(2, f(2)),
    ?assertEqual(3, f(3)).
```

...will stop at the first failure

```
====================== EUnit ======================
directory "ebin"
  fib: f_test (module 'fib')...*failed*
::error:{assertEqual_failed,[{module,fib},
                           {line,13},
                           {expression,"f ( 2 )"},
                           {expected,2},
                           {value,1}]}
  in function fib:'-f_test/0-fun-2-'/1
  in call from fib:f_test/0
```

Using a generator function

```
%% File: src/fib.erl
-module(fib).
-export([f/1]).
-include("global.hrl").

f(0) -> 1;
f(1) -> 1;
f(N) when N > 1 -> f(N-1) * f(N-2).

f_test_() ->
    [?_assertEqual(1, f(0)),
     ?_assertEqual(1, f(1)),
     ?_assertEqual(2, f(2)),
     ?_assertEqual(3, f(3))].
```

...creates a set of separate tests

```
    fib:11: f_test_...ok
    fib:12: f_test_...ok
    fib:13: f_test_...*failed*
::error:{assertEqual_failed,[{module,fib},
                             {line,13},
                             {expression,"f ( 2 )"},
                             {expected,2},
                             {value,1}]}
  in function fib:'-f_test_/0-fun-4-'/1
    fib:14: f_test_...*failed*
::error:{assertEqual_failed,[{module,fib},
                             {line,14},
                             {expression,"f ( 3 )"},
                             {expected,3},
                             {value,1}]}
  in function fib:'-f_test_/0-fun-6-'/1
```

Remember to test error cases

```erlang
%% File: src/fib.erl
-module(fib).
-export([f/1]).
-include("global.hrl").

f(0) -> 1;
f(1) -> 1;
f(N) when N > 1 -> f(N-1) + f(N-2).

f_test_() ->
  [?_assertEqual(1, f(0)),
   ?_assertEqual(1, f(1)),
   ?_assertEqual(2, f(2)),
   ?_assertError(function_clause, f(-1)),
   ?_assert(f(31) =:= 2178309)].
```

...and we should be good to go

```
======================= EUnit ========================
directory "ebin"
  module 'fib'
    fib:11: f_test_...ok
    fib:12: f_test_...ok
    fib:13: f_test_...ok
    fib:14: f_test_...ok
    fib:15: f_test_...[0.394 s] ok
    [done in 0.432 s]
=====================================================
  All 5 tests passed.
```

Tests in a separate module

```
%% File: src/fib_tests.erl
-module(fib_tests).
-include("global.hrl").

f_test_() ->
  [?_assertEqual(1, fib:f(0)),
   ?_assertEqual(1, fib:f(1)),
   ?_assertEqual(2, fib:f(2)),
   ?_assertError(function_clause, fib:f(-1)),
   ?_assert(fib:f(31) =:= 2178309)].
```

...which is used automatically

```
1> eunit:test(fib, [verbose]).
====================== EUnit ========================
module 'fib'
  module 'fib_tests'
    fib_tests:6: f_test_...ok
    fib_tests:7: f_test_...ok
    fib_tests:8: f_test_...ok
    fib_tests:9: f_test_...ok
    fib_tests:11: f_test_...[0.405 s] ok
    [done in 0.442 s]
  [done in 0.442 s]
===================================================
  All 5 tests passed.
```

An more efficient algorithm

```
%% File: src/fib.erl
-module(fib).
-export([f/1, g/1]).

f(0) -> 1;
f(1) -> 1;
f(N) when N > 1 -> f(N-1) + f(N-2).


g(N) when N >= 0 -> g(N, 0, 1).

g(0, _F1, F2) -> F2;
g(N,  F1, F2) -> g(N - 1, F2, F1 + F2).
```

...reference implementation test

```
%% File: src/fib_tests.erl
-module(fib_tests).
-include("global.hrl").

f_test_() ->
  [...].

g_test_() ->
    [?_assertError(function_clause, fib:g(-1)),
     ?_assertEqual(fib:f(0), fib:g(0)),
     ?_assertEqual(fib:f(1), fib:g(1)),
     ?_assertEqual(fib:f(2), fib:g(2)),
     ?_assertEqual(fib:f(17), fib:g(17)),
     ?_assertEqual(fib:f(31), fib:g(31))].
```

...new and old are equivalent

```
====================== EUnit ======================
module 'fib'
  module 'fib_tests'
    fib_tests:6: f_test_...ok
    fib_tests:7: f_test_...ok
    fib_tests:8: f_test_...ok
    fib_tests:9: f_test_...ok
    fib_tests:11: f_test_...[0.397 s] ok
    fib_tests:14: g_test_...ok
    fib_tests:16: g_test_...ok
    fib_tests:17: g_test_...ok
    fib_tests:18: g_test_...ok
    fib_tests:19: g_test_...ok
    fib_tests:20: g_test_...[0.425 s] ok
===================================================
  All 11 tests passed.
```

Generating tests dynamically

```
%% File: src/fib_tests.erl
-module(fib_tests).
-include("global.hrl").

f_test_() ->
  [...].

g_test_() ->
    [?_assertEqual(fib:f(N), fib:g(N))
     || N <- lists:seq(0,33)].

g_error_test() ->
    ?assertError(function_clause, fib:g(-1)).
```

...can take some time, though

```
    fib_tests:17: g_test_...ok
    fib_tests:17: g_test_...ok
    ...
    fib_tests:17: g_test_...[0.001 s] ok
    fib_tests:17: g_test_...[0.002 s] ok
    fib_tests:17: g_test_...[0.003 s] ok
    fib_tests:17: g_test_...[0.005 s] ok
    fib_tests:17: g_test_...[0.008 s] ok
    fib_tests:17: g_test_...[0.013 s] ok
    fib_tests:17: g_test_...[0.021 s] ok
    ...
    fib_tests:17: g_test_...[0.566 s] ok
    fib_tests:17: g_test_...[0.939 s] ok
    [done in 3.148 s]
==========================================================
  All 40 tests passed.
```

Don't overdo exhaustive testing

- Takes time (and the time tends to add up)
- Often, it does not give any more guarantees
- Try to cover the “interesting” cases first of all
  - Domain boundaries, zero, one, two, minus one
  - Empty list, ordered list, reverse-ordered list, ...
- A smaller number of randomly chosen cases could be a useful approach (but try to avoid using the same random seed every time).

Let's see if multicore is any help

```
%% File: src/fib_tests.erl
-module(fib_tests).
-include("global.hrl").

...

g_test_() ->
    {inparallel,
     [?_assertEqual(fib:f(N), fib:g(N))
      || N <- lists:seq(0,33)]
    }.

...
```

...not too bad, actually

```
    fib_tests:17: g_test_...ok
    fib_tests:17: g_test_...ok
    ...
    fib_tests:17: g_test_...[0.001 s] ok
    fib_tests:17: g_test_...ok
    fib_tests:17: g_test_...[0.003 s] ok
    fib_tests:17: g_test_...[0.011 s] ok
    fib_tests:17: g_test_...[0.015 s] ok
    fib_tests:17: g_test_...[0.023 s] ok
    fib_tests:17: g_test_...[0.036 s] ok
    ...
    fib_tests:17: g_test_...[1.378 s] ok
    fib_tests:17: g_test_...[1.085 s] ok
    [done in 1.853 s]
==========================================================
  All 40 tests passed.
```

A small server process with state

```
%% File: src/adder.erl
-module(adder).
-export([start/0, stop/1, add/2]).
start() -> spawn(fun server/0).
stop(Pid) -> Pid ! stop.
add(D, Pid) ->
  Pid ! {add, D, self()},
  receive {adder, N} -> N end.

server() -> server(0).
server(N) ->
  receive
    {add, D, From} ->
        From ! {adder, N + D},  server(N + D);
    stop -> ok
  end.
```

Setup and cleanup (“fixtures”)

```erlang
%% File: src/adder_tests.erl
-module(adder_tests).
-include("global.hrl").

named_test_() ->
  {setup,
   fun()-> P=adder:start(), register(srv, P), P end,
   fun adder:stop/1,
   [?_assertEqual(0, adder:add(0, srv)),
    ?_assertEqual(1, adder:add(1, srv)),
    ?_assertEqual(11, adder:add(10, srv)),
    ?_assertEqual(6, adder:add(-5, srv)),
    ?_assertEqual(-5, adder:add(-11, srv)),
    ?_assertEqual(0, adder:add(-adder:add(0, srv),
                                           srv))]
  }.
```

Instantiation of a bunch of tests

```
...

anonymous_test_() ->
  {setup, fun adder:start/0, fun adder:stop/1,
   fun (Srv) ->
    {inorder,    %% ensure that these run in order
     [?_assertEqual(0, adder:add(0, Srv)),
      ?_assertEqual(1, adder:add(1, Srv)),
      ?_assertEqual(11, adder:add(10, Srv)),
      ?_assertEqual(6, adder:add(-5, Srv)),
      ?_assertEqual(-5, adder:add(-11, Srv)),
      ?_assertEqual(0, adder:add(-adder:add(0, Srv),
                                 Srv))]
    }
   end}.
```

Single test with several asserts

```
anonymous_test_() ->
  {setup, fun adder:start/0, fun adder:stop/1,
   fun (Srv) ->
    [?_test(
      begin
       ?assertEqual(0, adder:add(0, Srv)),
       ?assertEqual(1, adder:add(1, Srv)),
       ?assertEqual(11, adder:add(10, Srv)),
       ?assertEqual(6, adder:add(-5, Srv)),
       ?assertEqual(-5, adder:add(-11, Srv)),
       ?assertEqual(0, adder:add(-adder:add(0, Srv),
                                 Srv))
      end)]
   end}.
```

Reduce copy-and-paste in tests

```
anonymous_test_() ->
  {setup, fun adder:start/0, fun adder:stop/1,
   fun (Srv) ->
    [?_test(
      begin
       assert_add(  0,  0, Srv),
       ...
       assert_add(-11, -5, Srv),
       assert_add(-adder:add(0, Srv), 0, Srv)
      end
     )]
   end}.

assert_add(D, N, Srv) ->
  ?assertEqual(N, adder:add(D, Srv)).
```

Break out subtests as functions

```
anonymous_test_() ->
  {setup, fun adder:start/0, fun adder:stop/1,
   fun (Srv) ->
    {with, Srv,
      [fun first_subtest/1, ...]
    }
   end}.

first_subtest(Srv) ->
  assert_add(  0,  0, Srv),
  ...
  assert_add(-11, -5, Srv),
  assert_add(-adder:add(0, Srv), 0, Srv).

assert_add(D, N, Srv) ->
  ?assertEqual(N, adder:add(D, Srv)).
```

'with' as body of setup

```
anonymous_test_() ->
  {setup, fun adder:start/0, fun adder:stop/1,
    {with,
      [fun first_subtest/1,
       ...]
    }
  }.

first_subtest(Srv) ->
  assert_add(  0,  0, Srv),
  ...
  assert_add(-11, -5, Srv),
  assert_add(-adder:add(0, Srv), 0, Srv).

assert_add(D, N, Srv) ->
  ?assertEqual(N, adder:add(D, Srv)).
```

Test code should not be sloppy

- Treat your test code like your normal code
  - Duplication is a bad smell
  - Try to say things at most once
  - Refactor your test code
- Remember that you can use all of Erlang
  - Break out repeated stuff into help functions
  - Use -ifdef(TEST) to conditionally compile help code
- Read the EUnit manual
  - There are many features that can assist you

That's it