Unit testing with EUnit

Richard Carlsson

Unit Testing in a Nutshell

- A "unit" can be any well-defined component
  - Function, Module, Library/API, Application, ...
- Tests the actual behaviour of program units
  - Each single test should try to check just one thing
  - A single test can either *pass* or *fail*
  - Check behaviour according to specification/docs
- A bunch of tests together make up a "test suite"

Eunit in action (the old fib/1)

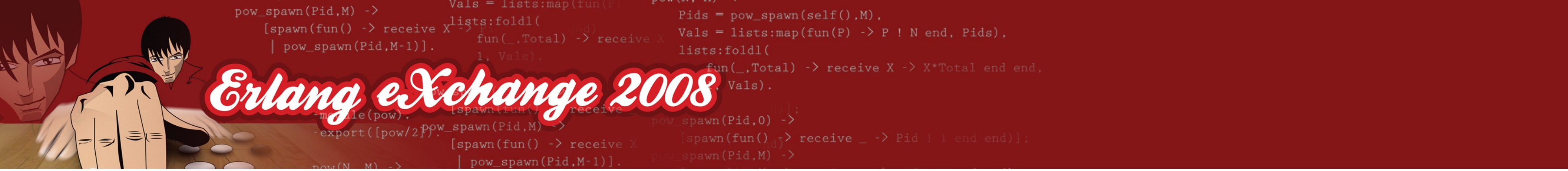
```
-module(fib).
-export([fib/1]).

fib(0) -> 1;
fib(1) -> 1.
```

Step one: include eunit.hrl

```
-module(fib).
-export([fib/1]).
-include_lib("eunit/include/eunit.hrl").

fib(0) -> 1;
fib(1) -> 1.
```

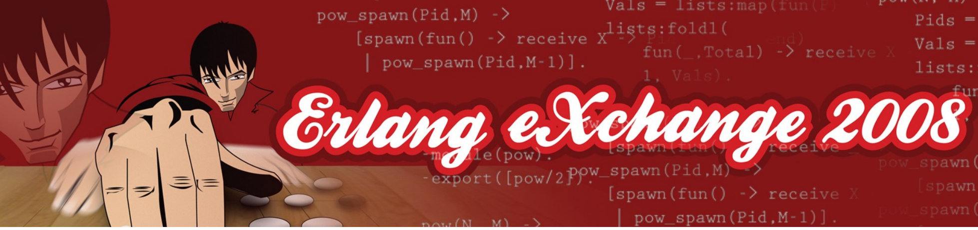


Naming conventions identify tests

```
-module(fib).
-export([fib/1]).
-include_lib("eunit/include/eunit.hrl").

fib(0) -> 1;
fib(1) -> 1.

fib0_test() -> ?assert(fib(0) == 1).
```



Test functions are auto-exported

```
-module(fib).
-export([fib/1]).
-include_lib(“eunit/include/eunit.hrl”).

fib(0) -> 1;
fib(1) -> 1.

fib0_test() -> ?assert(fib(0) == 1).

fib1_test() -> ?assert(fib(1) == 1).
```

Compiling and running

```
1> c(fib).
{ok,fib}
2>
```

The automatic test() function

```
1> c(fib).
{ok,fib}
2> fib:test().
```

The automatic test() function

```
1> c(fib).
{ok,fib}
2> fib:test().
  All 2 tests successful
ok
3>
```

Adding tests to drive development

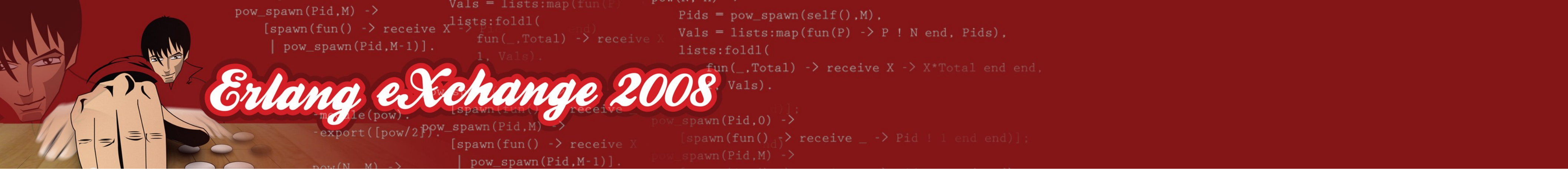
```
-module(fib).
-export([fib/1]).
-include_lib("eunit/include/eunit.hrl").

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

fib0_test() -> ?assert(fib(0) == 1).

fib1_test() -> ?assert(fib(1) == 1).

fib2_test() -> ?assert(fib(2) == 2).
```



Compact code with generators

```
-module(fib).
-export([fib/1]).
-include_lib("eunit/include/eunit.hrl").

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

fib_test_() ->
     [?_assert(fib(0) == 1),
      ?_assert(fib(1) == 1),
      ?_assert(fib(2) == 2)].
```

Compact code with generators

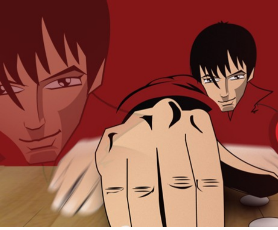
```
-module(fib).
-export([fib/1]).
-include_lib(“eunit/include/eunit.hrl”).

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

fib_test_() ->
    [?_assert(fib(0) == 1),
     ?_assert(fib(1) == 1),
     ?_assert(fib(2) == 2),
     ?_assert(fib(3) == 3),
     ?_assert(fib(4) == 5),
     ?_assert(fib(5) == 8)].
```

Using eunit:test() to run tests

```
3> c(fib).
{ok,fib}
4> eunit:test(fib).
```



Reading error reports

```
3> c(fib).
{ok,fib}
4> eunit:test(fib).
fib:12:fib_test_...*failed*
::{error,{assertion_failed,
          [{module, fib},
           {line,12},
           {expression, "fib ( 2 ) == 2"},
           {expected, true},
           {value, false}]},
...
...
=================================================
  Failed: 4  Aborted: 0  Skipped: 0  Succeeded: 2
5>
```

Fixing errors

```
-module(fib).
-export([fib/1]).
-include_lib("eunit/include/eunit.hrl").

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

fib_test_() ->
     [?_assert(fib(0) == 1),
      ?_assert(fib(1) == 1),
      ?_assert(fib(2) == 2),
      ?_assert(fib(3) == 3),
      ?_assert(fib(4) == 5),
      ?_assert(fib(5) == 8)].
```

Fixing errors

```
-module(fib).
-export([fib/1]).
-include_lib("eunit/include/eunit.hrl").

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

fib_test_() ->
    [?_assert(fib(0) == 1),
     ?_assert(fib(1) == 1),
     ?_assert(fib(2) == 2),
     ?_assert(fib(3) == 3),
     ?_assert(fib(4) == 5),
     ?_assert(fib(5) == 8)].
```

Also test the error cases

```
-module(fib).
-export([fib/1]).
-include_lib("eunit/include/eunit.hrl").

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

fib_test_() ->
     [?_assert(fib(0) == 1),
      ?_assert(fib(1) == 1),
      ?_assert(fib(2) == 2),
      ?_assert(fib(3) == 3),
      ?_assert(fib(4) == 5),
      ?_assert(fib(5) == 8),
      ?_assertError(function_clause, fib(-1))].
```

All done

```
5> c(fib).
{ok,fib}
6> eunit:test(fib).
  All 7 tests successful
ok
7>
```

Motivation

"But I'm a very good programmer – why should I spend my valuable time writing little trivial tests?"

Let unit tests help you save time

- Avoid undetected regressions
  - Changes of strategy during initial development
  - Bug fixes, years later and/or by someone else
  - Refactoring/rewriting to add new features
- The tests are “early adopters” of your code
  - Discover dependencies that you didn't think about
  - Shake out poor API design before it is too late
- You get usage examples for free

Things to test

- Sanity check computed values
  - Boundary cases ('reverse' on an empty list, etc.)
  - Things that the compiler or Dialyzer cannot detect (does 'reverse' really reverse the input?)
- Behaviour in case of errors or bad input
  - Ensure that it matches the documentation
- Resource management
  - Is everything cleaned up, even after a crash?

More things to test

- Process interaction
  - What happens if 100 processes try to run your code at the same time? Are there race conditions?
  - Do client processes hang if the server is killed?
  - Does your code actually work in the timeout cases?
- Assumptions about third party code
  - Write test cases for obscure or undocumented behaviour that you are relying on

EUnit: Functional unit testing

- The "xUnit family" of frameworks (JUnit, etc.) are mostly built on object-oriented principles
  - Heavy use of inheritance:
    - scaffolding (code for running the tests)
    - setup/teardown of test contexts (open/close files, etc.)
- We want to be able to handle tests as data
  - Lambda expressions (funs) make natural test cases
  - Represent test sets as collections (lists) of funs
  - Deep lists make it easy to combine test sets

Test case conventions

- A test case is represented as a fun that takes no arguments
- A test fails if evaluation throws an exception
- Any return value is considered a success

```
Test = fun () -> ... end,

try Test() of
    _ -> pass
catch
    _:_ -> fail
end
```

Running tests

- Most basic usage:

```
eunit:test(TestSet)
```

- Where `TestSet` can be:
  - a fun (taking zero arguments)
  - a module name
  - various other things (as we shall see)
  - a (deep) list of funs, module names, and other things

Modules as test sets

- Given a module name (any atom), EUnit will look for exported functions with special names:

```
..._test()      % simple test
..._test_()     % test generator
```

- Simple test functions are treated just as funs: they represent a single test case.
- Test generators are *called immediately*, and should *return* a test set (a fun, module name, list of tests, etc.)

EUnit utility macros

- `-include_lib("eunit/include/eunit.hrl").`
- Make the code more compact and readable
- Generate more informative exceptions

```
?assert(1 + 1 == 2)
?assertMatch({foo, _}, make_foo(42))
?assertError(badarith, 1/0)
?assertThrow(ouch, throw(ouch))
```

Test object macros

- Macros whose names begin with "_" wrap their arguments in a fun, creating a test object:

```
?_test(Expr)  <=>  fun()-> Expr end
```

- All assert macros have _-forms:

```
?_test(?assert(BoolExpr))
 <=> ?_assert(BoolExpr)
```

- Usage comparison:

```
simple_test()-> ?assert(BoolExpr).
fun_test_()-> ?_assert(BoolExpr).
```

Advanced test descriptors

- Labels

```
{"Label Text", Tests}
```

- Testing applications or directories

```
{dir, "Directory Path"}
{application, AppName}
```

- Timeouts

```
{timeout, Seconds, Tests}
```

- Running tests in subprocesses

```
{spawn, Tests}
{spawn, Node, Tests}
```

More advanced test descriptors

- Specifying the execution order

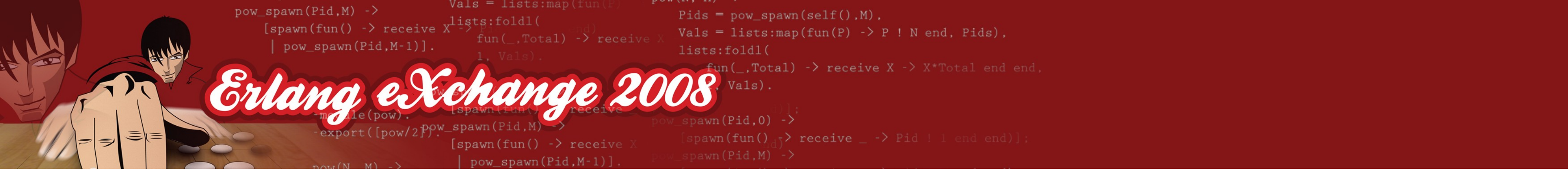
```
{inparallel, Tests}
{inorder, Tests}
```

- Generators

```
{generator, Fun}
{generator, Module, Function}
```

- Test setup and cleanup (fixtures)

```
{setup, SetupFun,          % () -> R::any()
        CleanupFun,        % (R::any()) -> any()
        (Instantiator      % (R::any()) -> Tests
         | Tests) }
```



And some even more advanced

- Repeated setup and cleanup

```
{foreach, SetupFun,
          CleanupFun,
          [ Instantiator|Tests ] }

{foreachx, SetupFunX,
           CleanupFunX,
           [ {X::any(), XInstantiator} ] }
```

- Instantiating (distributing over) a set of tests

```
{with, X::any(),
 [ AbstractTestFun::((any()) -> any()) ] }
```

Test-driven design

- Write unit tests (and run them, often) while developing the program, not afterwards
- Write tests for a feature *before* you start implementing it (work on minimal “features”)
- Move on immediately when all tests pass
- Good for focus and productivity:
  - Concentrate on solving the right problems
  - Avoid overspecification and premature optimization

Triangulation

- Start with the simplest possible properties
  - Usually things like boundary values
- Implement the simplest possible solutions
  - E.g., just hard-code some known return values
  - Ensures that the most basic tests are in place before you start writing any complicated code
- Don't move on until you have tests to force it
  - Ensures that you are testing nontrivial properties

Some final tips

- Be sure that the test can actually fail
  - Tests that cannot fail are useless
  - Insert a bug to trigger it (and remove it again)
- Keep a plaintext TODO-list of ideas for tests
  - Helps you focus on the test you're working on
- Your tests are also part of your code base
  - Don't lower your standards (too much)
  - Avoid copy-pasting tests; refactor when you can