Coding standards – Erlang.

This document is meant as a guideline for code practices for using Erlang. It's not intended to tell you how to write everything but merely suggests a few best practices and techniques to use so that the code is easier to interpret and maintain.

We are using some of the standards and practices found in the following document:

http://www.erlang.se/doc/programming_rules.shtml

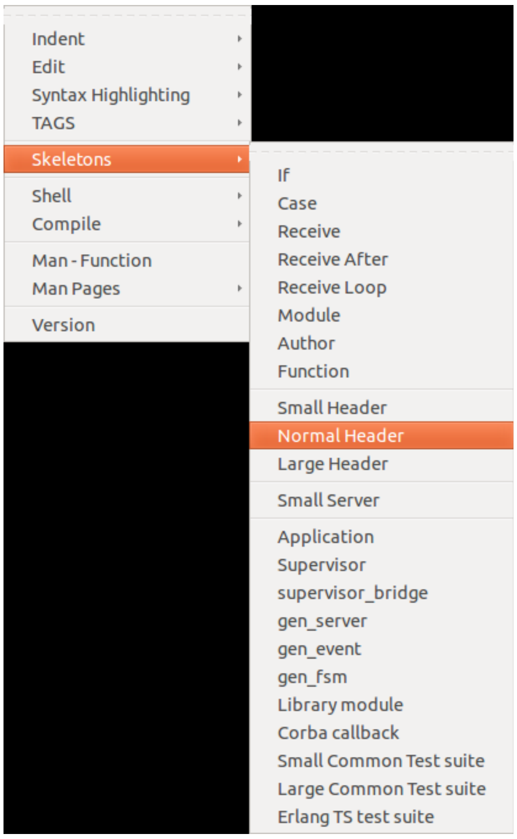
Table of Contents

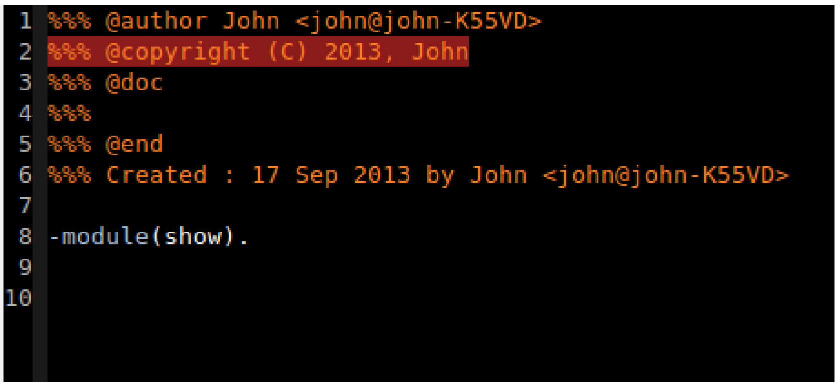
1 File headers.

Ideally all modules and most functions should have a concise heading that describes what the module / function does and how it's possibly combined with other functions.

If you're using emacs and have the Erlang language options install, simply click the following:

Erlang > Skeletons > *Header type you require* (See below for an example).





```
%%% @author John <john@john-K55VD>
%%% @copyright (C) 2013, John
%%% @doc
%%%
%%% @end
%%% Created : 17 Sep 2013 by John <john@john-K55VD>

-module(show).

```

Full size file headers are unnecessary as they only add another 2 lines.

If you're not using emacs, check your appropriate text editor / IDE's documentation / plug-ins to see if there's a similar plug in that allows you to create skeletons. Failing that, you'll have to type them in manually. One is here in case of such a scenario:

%%% @author NAME <NAME@DEVICE>

%%% @copyright (C) YEAR, NAME

%%% @doc

%%%

%%% @end

%%% Created : DATE by NAME <NAME@DEVICE>

1.1 *Revision History.*

Each new revision of a given module or function must have its revision history updated to show who has worked on it, and what was added. An example can be seen below:

```
%%%---------------------------------------------------------------------
%%% Revision History
%%%---------------------------------------------------------------------
%%% Rev PA1 Date 960230 Author Fred Bloggs (ETXXXXX)
%%% Intitial pre release. Functions for adding and deleting foobars
%%% are incomplete
%%%---------------------------------------------------------------------
%%% Rev A Date 960230 Author Johanna Johansson (ETXYYY)
%%% Added functions for adding and deleting foobars and changed
%%% data structures of foobars to allow for the needs of the Baz
%%% signalling system
%%%---------------------------------------------------------------------
```

1.2 Header Description.

File headers should start with a short description of what the module contains, does and what it exports. An example can be seen below:

```
%%%---------------------------------------------------------------------
%%% Description module foobar_data_manipulation
%%%---------------------------------------------------------------------
%%% Foobars are the basic elements in the Baz signalling system. The
%%% functions below are for manipulating that data of foobars and for
%%% etc etc etc
%%%---------------------------------------------------------------------
%%% Exports
%%%---------------------------------------------------------------------
%%% create_foobar(Parent, Type)
%%%   returns a new foobar object
%%%   etc etc etc
%%%---------------------------------------------------------------------
```

Note: If there are any weakness, bugs, or just dodgy code, please make an extra comment either in the modules definition or around the code in question, so that it can be clearly seen. This goes for parts of a module that are incomplete.

1.3 Copyright Information.

The Erlang code standards documentation suggests that we have a copyright header included with the modules, I think for our project this is a little unnecessary as we're most likely not going to be publishing the code for commercial use. Nevertheless an example is below:

```
%%%---------------------------------------------------------------------
%%% Copyright Ericsson Telecom AB 1996
%%%
%%% All rights reserved. No part of this computer programs(s) may be
%%% used, reproduced,stored in any retrieval system, or transmitted,
%%% in any form or by any means, electronic, mechanical, photocopying,
%%% recording, or otherwise without prior written permission of
%%% Ericsson Telecom AB.
%%%---------------------------------------------------------------------
```

It's also important to note that if we're using code from other sources, that we should properly attribute the code to whoever created it, it's not cool to steal code!

2 Comments.

This section will give some basic advice for commenting your code, Erlang has some conventions that we should follow.

2.1 Basic Comments.

Comments should be clear and concise without fluffing up the comments with unneeded wording. Comments should be kept up to date with the code, this should be pretty straight forward that if you update a module and invalidate a comment, you should change that comment to reflect that change.

Comments about modules should be unindented and starting with 3 percentage characters (%%%).

Comments about functions should start without indentation and should be proceeded by 2 percentage characters (%%).

Standard comments should be proceeded by a single percentage character (%). Place comments above the lines to which they refer or if possible, following the line to which they refer.

An example can be seen below:

```
%% Comment about function
some_useful_functions(UsefulArgugument) ->
  another_functions(UsefulArgugument),    % Comment at end of line
  % Comment about complicated_stmnt at the same level of indentation
  complicated_stmnt,
```

2.2 *Function Comments.*

When commenting functions, there a few Erlang practices to keep in mind, the function comment should:

- Show the purpose of the function
- The domain of valid inputs, the data structures of the arguments to their functions with their meanings.
- The domain of the output function, the data structures of the return value to their functions with their meanings.
- If the function implements a complex algorithm, it should be explained.
- If exit/1 or throw/1 are used, the causes of their failures / exit signals should be given.
- Any side effects of the function should be clearly conveyed.

An example of this is shown below:

```
%%---------------------------------------------------------------------
%% Function: get_server_statistics/2
%% Purpose: Get various information from a process.
%% Args:   Option is normal|all.
%% Returns: A list of {Key, Value}
%%     or {error, Reason} (if the process is dead)
%%---------------------------------------------------------------------
get_server_statistics(Option, Pid) when pid(Pid) ->
```

2.3 Data Structures.

We're no doubt going to be using custom data structures in the project, and as such we should be using proper commenting style when defining them. Data structures should come with a plain text description, see below for an example:

```
%% File: my_data_structures.h

%%------------------------------------------------------------------
%% Data Type: person
%% where:
%%    name: A string (default is undefined).
%%    age: An integer (default is undefined).
%%    phone: A list of integers (default is []).
%%    dict:     A dictionary containing various information about the person.
%%       A {Key, Value} list (default is the empty list).
%%------------------------------------------------------------------
-record(person, {name, age, phone = [], dict = []}).
```

As seen above, we simply write out in plain text what data types are required for the data structure / record.

3 Styling Conventions.

As with all programming languages, there are styling conventions that are considered standard, we should therefore be using them in our code.

3.1 Code length / Nesting.

This is usually a standard in most languages. The specification says that there should ideally be no more than 2 levels of indentation. As we're not (or shouldn't really be) using if statements this leaves cases. In this respect, they consider nested cases to be poor practice, so please avoid doing this.

3.2 Variable Names.

Choose meaningful names, this can be a problem when you've used up a lot of your “sensible” names already in other areas of the module / function. Use a standard where you have capitals at the start of names, or underlines. Such as:

my_variable,

or

MyVariable

When using “_” as a don't care variable, try not to use it completely on it's on, it can be used at the start of a variable which can be useful to others reading code that might not want to know what you're not caring about. Just make sure to put it at the start of a variable.

_A, _B

_MyVariable

3.3 Function Names.

Function names should agree with what the function is doing. It should return the same types of arguments implied by it's name. Use conventional names where possible, i.e. start, stop, pause, init, main_loop etc.

Functions in different modules that solve the same problems should be named the same: Module:module_info().

Some prefixes can be helpful for certain functions, such as when writing your own boolean (true / false) functions, starting them with _is might be a good indicator of their intention.

3.4 Module Names.

Erlang has a flat model structure, we never get modules within modules however, it's often a good idea to simulate the idea of an hierarchy within the modules. This can be done by giving modules the same prefix, such as:

DB_init

DB_search

DB_close

etc.

3.5 General Formatting.

We all probably have different ways of coding. To minimise problems that can occur as a result of this, we should use a standardised way of writing certain parts of code.

For example, try to remove white space where possible,

i.e. in tuples {A,B,C} instead of {A, B, C},

If we can keep a good standard of code, it should make maintainability and readability better.

4 <u>Specific Erlang Conventions.</u>

Erlang has some more specific conventions that are not necessarily apparent at a first glance. This section will list some of the most specific and most relevant to our project. There are more which are not mentioned here, if you want to see for yourself what they are, please visit:

http://www.erlang.se/doc/programming_rules.shtml#REF11301

4.1 Use records.

We should be using records as our primary data structures. It should be a tagged tuple. Similar in construction as a struct in C\C++ or a record in pascal.

If the record is to be used in several modules, it should be placed into a header file (With the extension of .hrl) that is included in the from the modules. If the record is only used in a single module, the definition should be at the start of that module.

4.2 Selectors and Constructors.

We should use selectors and constructors provided by the record feature to manage instances of the records that we create. We should not use matching that assumes that the record is a tuple. An example is:

```
demo() ->
  P = #person{name = "Joe", age = 29},
  #person{name = Name1} = P,% Use matching, or...
  Name2 = P#person.name. % use the selector like this.
```

Don't program like this:

```
demo() ->
  P = #person{name = "Joe", age = 29},
  {person, Name, _Age, _Phone, _Misc} = P. % Don't do this
```

This should become clearer once we have covered records / selectors / constructors.

4.3 Returning Values.

We should not be returning values untagged, for example:

Don't program like this:

```
keysearch(Key, [{Key, Value}|_Tail]) ->
  Value; %% Don't return untagged values!
keysearch(Key, [{_WrongKey, _WrongValue} | Tail]) ->
  keysearch(Key, Tail);
keysearch(Key, []) ->
  false.
```

Then the {Key, Value} cannot contain the false value. This is the correct solution:

```
keysearch(Key, [{Key, Value}|_Tail]) ->
  {value, Value}; %% Correct. Return a tagged value.
keysearch(Key, [{_WrongKey, _WrongValue} | Tail]) ->
  keysearch(Key, Tail);
keysearch(Key, []) ->
  false.
```

Again, this should become clearer once we use records more often.

4.4 Use of Import.

The programming rules document suggests that we do not use the import function, it makes code harder to read and also makes it harder to determine where modules are defined. Instead we should be using the exref (Cross reference tool) to point to where the modules are.

4.5 Use of Export.

A function should be exported for some of the following reasons:

1. It is a user interface to a module
2. Is an interface function to other modules

3. It is called from apply, spawn etc But only within its module.

Use different -export groupings and comment them separately, see below for an example:

```
%% user interface
-export([help/0, start/0, stop/0, info/1]).

%% intermodule exports
-export([make_pid/1, make_pid/3]).
-export([process_abbrevs/0, print_info/5]).

%% exports for use within module only
-export([init/1, info_log_impl/1]).
```

4.6 Use of Catch and Throw.

5 Other Noteworthy Information.

I've not covered everything that's in the documentation, this document was not supposed to be a carbon copy. If you want to find out any more information, please refer to the web address at the top of this document. If you feel that I've missed anything crucial, please let me know and I'll add it here if possible.

For now, I'll reserve this section for small additions or notes that we deem needed.

5.1 Download / Internet Links.

It might be useful to consolidate the links for downloads etc. in the same place.

Emacs: http://www.gnu.org/software/emacs/

Erlang Mode: http://www.erlang.org/doc/apps/tools/erlang_mode_chapter.html

Practices Document: http://www.erlang.se/doc/programming_rules.shtml