Program Development Using Erlang - Programming Rules and Conventions

Abstract

This is a description of programming rules and advise for how to write systems using Erlang.

Note

This document is a preliminary document and is not complete

The requirements for the use of EBC's "Base System" are not documented here, but must be followed at a very early design phase if the "Base System" is to be used. These requirements are documented in 1/10268-AND 10406 Uen "MAP - Start and Error Recovery"

Contents

1 Purpose

This paper lists some aspects which should be taken into consideration when specifying and programming software systems using Erlang. It does not attempt to give a complete description of general specification and design activities which are independent of the use of Erlang.

2 Structure and Erlang Terminology

Erlang systems are divided into **modules**. Modules are composed of **functions** and **attributes**. Functions are either only visible inside a module or they are **exported** i.e. they can also be called by other functions in other modules. Attributes begin with "`-`" and are placed in the beginning of a module.

The *work* in a system designed using Erlang is done by **processes**. A process is a *job* which can use functions in many modules. Processes communicate with each other by **sending messages**. Processes **receive** messages which are sent to them, a process can decide which messages it is prepared to receive. Other messages are queued until the receiving process is prepared to receive them.

A process can supervise the existence of another process by setting up a **link** to it. When a process terminates, it automatically sends **exit signals** to the process to which it is linked. The default behavior of a process receiving an exit signal is to terminate and to propagate the signal to its linked processes. A process can change this default behavior by **trapping exits**, this causes all exit signals sent to a process to be turned into messages.

A **pure function** is a function that returns the same value given the same arguments regardless of the context of the call of the function. This is what we normally expect from a mathematical function. A function that is not pure is said to have **side effects**.

Side effects typically occur if a function a) sends a message b) receives a message c) calls `exit` d) calls any BIF which changes a processes environment or mode of operation (e.g. get/1, put/2, erase/1, process_flag/2 etc).

Warning: This document contains examples of bad code. Such examples will be written using this ugly font.

3 SW Engineering Principles

3.1 Export as few functions as possible from a module

Modules are the basic code structuring entity in Erlang. A module can contain a large number of functions but only functions which are included in the export list of the module can be called from outside the module.

Seen from the outside the complexity of a module depends upon the number of functions which are exported from the module. A module which exports one or two functions is usually easier to understand than a module which exports dozens of functions.

Modules where the ratio of exported/non-exported functions is low are desirable in that a user of the module only needs to understand the functionality of the functions which are exported from the module.

In addition, the writer or maintainer of the code in the module can change the internal structure of the module in any appropriate manner provided the external interface remains unchanged.

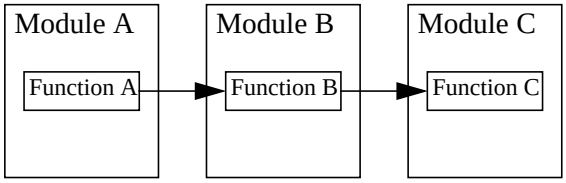
3.2 Try to reduce intermodule dependencies

A module which calls functions in many different modules will be more difficult to maintain than a module which only calls functions in a few different modules.

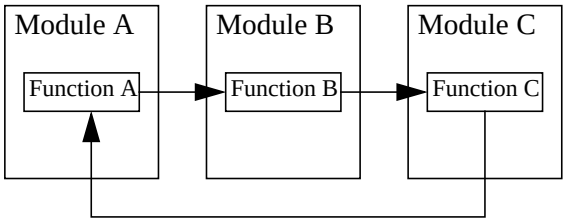
This is because each time we make a change to a module interface, we have to check all places in the code where this module is called. Reducing the interdependencies between modules simplifies the problem of maintaining these modules.

We can simplify the system structure by reducing the number of different modules which are called from a given module.

Note also that it is desirable that the inter-module calling dependencies form a tree and not a cyclic graph. Example:



But not



3.3 Put commonly used code into libraries

Commonly used code should be placed into libraries. The libraries should be collections of related functions. Great effort should be made in ensuring that libraries contain functions of the same type. Thus a library such as `lists` containing only functions for manipulating lists is a good choice, whereas a library, `lists_and_maths` containing a combination of functions for manipulating lists and for mathematics is a very bad choice.

The best library functions have no side effects. Libraries with functions with side effects limit the re-usability.

3.4 Isolate “tricky” or “dirty” code into separate modules

Often a problem can be solved by using a mixture of clean and dirty code. Separate the clean and dirty code into separate modules.

Dirty code is code that does dirty things. Example:

- Uses the process dictionary.
- Uses `erlang:process_info/1` for strange purposes.
- Does anything that you are not supposed to do (but have to do).

Concentrate on trying to maximize the amount of clean code and minimize the amount of dirty code. Isolate the dirty code and clearly comment or otherwise document all side effects and problems associated with this part of the code.

3.5 Don’t make assumptions about what the caller will do with the results of a function

Don’t make assumptions about why a function has been called or about what the caller of a function wishes to do with the results.

For example, suppose we call a routine with certain arguments which may be invalid. The implementer of the routine should not make any assumptions about what the caller of the function wishes to happen when the arguments are invalid.

Thus we should not write code like

```
do_something(Args) ->
    case check_args(Args) of
        ok ->
            {ok, do_it(Args)};
        {error, What} ->
            String = format_the_error(What),
            io:format("* error:~s\n", [String]), %% Don't do this
            error
    end.
```

Instead write something like:

```
do_something(Args) ->
   case check_args(Args) of
      ok ->
         {ok, do_it(Args)};
      {error, What} ->
         {error, What}
   end.

error_report({error, What}) ->
   format_the_error(What).
```

In the former case the error string is always printed on standard output, in the latter case an error descriptor is returned to the application. The application can now decide what to do with this error descriptor.

By calling `error_report/1` the application can convert the error descriptor to a printable string and print it if so required. But this may not be the desired behavior - in any case the decision as to what to do with the result is left to the caller.

3.6 Abstract out common patterns of code or behavior

Whenever you have the same pattern of code in two or more places in the code try to isolate this in a common function and call this function instead of having the code in two different places. Copied code requires much effort to maintain.

If you see similar patterns of code (i.e. almost identical) in two or more places in the code it is worth taking some time to see if one cannot change the problem slightly to make the different cases the same and then write a small amount of additional code to describe the differences between the two.

Avoid “copy” and “paste” programming, use functions!

3.7 Top-down

Write your program using the top-down fashion, not bottom-up (starting with details). Top-down is a nice way of successively approaching details of the implementation, ending up with defining primitive functions. The code will be independent of representation since the representation is not known when the higher levels of code are designed.

3.8 Don’t optimize code

Don’t optimize your code at the first stage. First make it right, then (if necessary) make it fast (while keeping it right).

3.9 Use the principle of “least astonishment”

The system should always respond in a manner which causes the “least astonishment” to the user - i.e. a user should be able to predict what will happen when they do something and not be astonished by the result.

This has to do with consistency, a consistent system where different modules do things in a similar manner will be much easier to understand than a system where each module does things in a different manner.

If you get astonished by what a function does, either your function solves the wrong problem or it has a wrong name.

3.10 Try to eliminate side effects

Erlang has several primitives which have side effects. Functions which use these *cannot be easily re-used* since they cause permanent changes to their environment and you have to know the exact state of the process before calling such routines.

Write as much as possible of the code with side-effect free code.

Maximize the number of pure functions.

Collect together the functions which have side effect and clearly document all the side effects.

With a little care most code can be written in a side-effect free manner - this will make the system a lot easier to maintain, test and understand.

3.11 Don't allow private data structure to "leak" out of a module

This is best illustrated by a simple example. We define a simple module called queue - to implement queues:

```
-module(queue).
-export([add/2, fetch/1]).

add(Item, Q) ->
    lists:append(Q, [Item]).

fetch([H|T]) ->
    {ok, H, T};
fetch([]) ->
    empty.
```

This implements a queue as a list, unfortunately to use this the user must know that the queue is represented as a list. A typical program to use this might contain the following code fragment:

```
NewQ = [], % Don't do this
Queue1 = queue:add(joe, NewQ),
Queue2 = queue:add(mike, Queue1), ....
```

This is bad - since the user a) needs to know that the queue is represented as a list and b) the implementer cannot change the internal representation of the queue (this they might want to do later to provide a better version of the module).

Better is:

```
-module(queue).
-export([new/0, add/2, fetch/1]).

new() ->
   [].

add(Item, Q) ->
   lists:append(Q, [Item]).

fetch([H|T]) ->
   {ok, H, T};
fetch([]) ->
   empty.
```

Now we can write:

```
NewQ = queue:new(),
Queue1 = queue:add(joe, NewQ),
Queue2 = queue:add(mike, Queue1), ...
```

Which is much better and corrects this problem. Now suppose the user needs to know the length of the queue, they might be tempted to write:

```
Len = length(Queue) % Don't do this
```

since they know that the queue is represented as a list. Again this is bad programming practice and leads to code which is very difficult to maintain and understand. If they need to know the length of the queue then a length function must be added to the module, thus:

```
-module(queue).
-export([new/0, add/2, fetch/1, len/1]).

new() -> [].
```

```
add(Item, Q) ->
   lists:append(Q, [Item]).
fetch([H|T]) ->
{ok, H, T};
fetch([]) ->
   empty.
len(Q) ->
   length(Q).
```

Now the user can call `queue:len(Queue)` instead.

Here we say that we have “abstracted out” all the details of the queue (the queue is in fact what is called an “abstract data type”).

Why do we go to all this trouble? - the practice of abstracting out internal details of the implementation allows us to change the implementation without changing the code of the modules which call the functions in the module we have changed. So, for example, a better implementation of the queue is as follows:

```
-module(queue).
-export([new/0, add/2, fetch/1, len/1]).
new() ->
   {[],[]}.
add(Item, {X,Y}) -> % Faster addition of elements
   {[Item|X], Y}.
fetch({X, [H|T]}) ->
   {ok, H, {X,T}};
fetch({[], []) ->
   empty;
fetch({X, []) ->
   % Perform this heavy computation only sometimes.
   fetch({[],lists:reverse(X)}).
len({X,Y}) ->
   length(X) + length(Y).
```

3.12 Make code as deterministic as possible

A deterministic program is one which will always run in the same manner no matter how many times the program is run. A non-deterministic program may deliver different results each time it is run. For debugging purposes it is a good idea to make things as deterministic as possible. This helps make errors reproducible.

For example, suppose one process has to start five parallel processes and then check that they have started correctly, suppose further that the order in which these five are started does not matter.

We could then choose to either start all five in parallel and then check that they have all started correctly but it would be better to start them one at a time and check that each one has started correctly before starting the next one.

3.13 Do not program “defensively”

A defensive program is one where the programmer does not “trust” the input data to the part of the system they are programming. In general one should not test input data to functions for correctness. Most of the code in the system should be written with the assumption that the input data to the function in question is correct. Only a small part of the code should actually perform any checking of the data. This is usually done when data “enters” the system for the first time, once data has been checked as it enters the system it should thereafter be assumed correct.

Example:

```
%% Args: Option is all|normal
get_server_usage_info(Option, AsciiPid) ->
   Pid = list_to_pid(AsciiPid),
   case Option of
      all -> get_all_info(Pid);
      normal -> get_normal_info(Pid)
   end.
```

The function will crash if `Option` neither `normal` nor `all`, and it should do that. The caller is responsible for supplying correct input.

3.14 Isolate hardware interfaces with a device driver

Hardware should be isolated from the system through the use of device drivers. The device drivers should implement hardware interfaces which make the hardware appear as if they were Erlang processes. Hardware should be made to look and behave like normal Erlang processes. Hardware should appear to receive and send normal Erlang messages and should respond in the conventional manner when errors occur.

3.15 Do and undo things in the same function

Suppose we have a program which opens a file, does something with it and closes it later. This should be coded as:

```
do_something_with(File) ->
   case file:open(File, read) of,
      {ok, Stream} ->
        doit(Stream),
        file:close(Stream) % The correct solution
      Error -> Error
   end.
```

Note the symmetry in opening the file (`file:open`)and closing it (`file:close`) in the same routine. The solution below is much harder to follow and it is not obvious which file that is closed. Don't program it like this:

```
do_something_with(File) ->
case file:open(File, read) of,
        {ok, Stream} ->
              doit(Stream)
        Error -> Error
    end.

doit(Stream) ->
    ....,
    func234(...,Stream,...).

 ...

func234(..., Stream, ...) ->
    ...,
    file:close(Stream) %% Don't do this
```

4 Error Handling

4.1 Separate error handling and normal case code

Don’t clutter code for the “normal case” with code designed to handle exceptions. As far as possible you should only program the normal case. If the code for the normal case fails, your process should report the error and crash as soon as possible. Don’t try to fix up the error and continue. The error should be handled in a different process (See “Each process should only have one “role”” on page 15.).

Clean separation of error recovery code and normal case code should greatly simplify the overall system design.

The error logs which are generated when a software or hardware error is detected will be used at a later stage to diagnose and correct the error. A permanent record of any information that will be helpful in this process should be kept.

4.2 Identify the error kernel

One of the basic elements of system design is identifying which part of the system has to be correct and which part of the system does not have to be correct.

In conventional operating system design the kernel of the system is assumed to be, and must be, correct, whereas all user application programs do not necessarily have to be correct. If a user application program fails this will only concern the application where the failure occurred but should not affect the integrity of the system as a whole.

The first part of the system design must be to identify that part of the system which must be correct; we call this the error kernel. Often the error kernel has some kind of real-time memory resident data base which stores the state of the hardware.

5 Processes, Servers and Messages

5.1 Implement a process in one module

Code for implementing a single process should be contained in one module. A process can call functions in any library routines but the code for the “top loop” of the process should be contained in a single module. The code for the top loop of a process should not be split into several modules - this would make the flow of control extremely difficult to understand. This does not mean that one should not make use of generic server libraries, these are for helping structuring the control flow.

Conversely, code for no more than one kind of process should be implemented in a single module. Modules containing code for several different processes can be extremely difficult to understand. The code for each individual process should be broken out into a separate module.

5.2 Use processes for structuring the system

Processes are the basic system structuring elements. But don’t use processes and message passing when a function call can be used instead.

5.3 Registered processes

Registered processes should be registered with the same name as the module. This makes it easy to find the code for a process.

Only register processes that should live a long time.

5.4 Assign exactly one parallel process to each true concurrent activity in the system

When deciding whether to implement things using sequential or parallel processes then the structure implied by the intrinsic structure of the problem should be used. The main rule is:

“Use one parallel process to model each truly concurrent activity in the real world”

If there is a one-to-one mapping between the number of parallel processes and the number of truly parallel activities in the real world, the program will be easy to understand.

5.5 Each process should only have one “role”

Processes can have different roles in the system, for example in the client-server model.

As far as possible a process should only have one role, i.e. it can be a client or a server but should not combine these roles.

Other roles which process might have are:

Supervisor: watches other processes and restarts them if they fail.
Worker: a normal work process (can have errors).
Trusted Worker: not allowed to have errors.

5.6 Use generic functions for servers and protocol handlers wherever possible

In many circumstances it is a good idea to use generic server programs such as the `generic` server implemented in the standard libraries. Consistent use of a small set of generic servers will greatly simplify the total system structure.

The same is possible for most of the protocol handling software in the system.

5.7 Tag messages

All messages should be tagged. This makes the order in the receive statement less important and the implementation of new messages easier.

Don’t program like this:

```
loop(State) ->
    receive
        ...
        {Mod, Funcs, Args} -> % Don’t do this
            apply(Mod, Funcs, Args},
            loop(State);
        ...
    end.
```

The new message `{get_status_info, From, Option}` will introduce a conflict if it is placed below the `{Mod, Func, Args}` message.

If messages are synchronous, the return message should be tagged with a new atom, describing the returned message. Example: if the incoming message is tagged `get_status_info`, the returned message could be tagged `status_info`. One reason for choosing different tags is to make debugging easier.

This is a good solution:

```
loop(State) ->
   receive
      ...
      {execute, Mod, Funcs, Args} -> % Use a tagged message.
         apply(Mod, Funcs, Args},
         loop(State);
      {get_status_info, From, Option} ->
         From ! {status_info, get_status_info(Option, State)},
         loop(State);
      ...
   end.
```

5.8 Flush unknown messages

Every server should have an `Other` alternative in at least one `receive` statement. This is to avoid filling up message queues. Example:

```
main_loop() ->
   receive
      {msg1, Msg1} ->
         ...,
         main_loop();
      {msg2, Msg2} ->
         ...,
         main_loop();
      Other -> % Flushes the message queue.
         error_logger:error_msg(
               "Error: Process ~w got unknown msg ~w~n.",
               [self(), Other]),
         main_loop()
   end.
```

5.9 Write tail-recursive servers

All servers *must* be tail-recursive, otherwise the server will consume memory until the system runs out of it.

Don’t program like this:

```
loop() ->
    receive
        {msg1, Msg1} ->
            ...,
            loop();
        stop ->
            true;
        Other ->
            error_logger:log({error, {process_got_other, self(), Other}}),
            loop()
    end,
    io:format("Server going down").        % Don’t do this!
                                           % This is NOT tail-recursive
```

This is a correct solution:

```
loop() ->
   receive
      {msg1, Msg1} ->
         ...,
         loop();
      stop ->
         io:format("Server going down");
      Other ->
         error_logger:log({error, {process_got_other, self(), Other}}),
         loop()
   end. % This is tail-recursive
```

If you use some kind of server library, for example `generic`, you automatically avoid doing this mistake.

5.10 Interface functions

Use functions for interfaces whenever possible, avoid sending messages directly. Encapsulate message passing into interface functions. There are cases where you can’t do this.

The message protocol is internal information and should be hidden to other modules.

Example of interface function:

```
-module(fileserver).

-export([start/0, stop/0, open_file/1, ...]).

open_file(FileName) ->
   fileserver ! {open_file_request, FileName},
   receive
      {open_file_response, Result} -> Result
   end.

...<code>...
```

5.11 Time-outs

Be careful when using `after` in `receive` statements. Make sure that you handle the case when the message arrives later (See “Flush unknown messages” on page 16.).

5.12 Trapping exits

As few processes as possible should trap exit signals. Processes should either trap exits or they should not. It is usually very bad practice for a process to “toggle” trapping exits.

6 Various Erlang Specific Conventions

6.1 Use records as the principle data structure

Use records as the principle data structure. A record is a tagged tuple and was introduced in Erlang version 4.3 and thereafter (see EPK/NP 95:034). It is similar to `struct` in C or `record` in Pascal.

If the record is to be used in several modules, its definition should be placed in a header file (with suffix .hrl) that is included from the modules. If the record is only used from within one module, the definition of the record should be in the beginning of the file the module is defined in.

The record features of Erlang can be used to ensure cross module consistency of data structures and should therefore be used by interface functions when passing data structures between modules.

6.2 Use selectors and constructors

Use selectors and constructors provided by the record feature for managing instances of records. Don't use matching that explicitly assumes that the record is a tuple. Example:

```
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
```

6.3 Use tagged return values

Use tagged return values.

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.
```

6.4 Use catch and throw with extreme care

Do not use `catch` and `throw` unless you know exactly what you are doing! Use `catch` and `throw` as little as possible.

`Catch` and `throw` can be useful when the program handles complicated and unreliable input (from the outside world, not from your own reliable program) that may cause errors in many places deeply within the code. One example is a compiler.

6.5 Use the process dictionary with extreme care

Do not use `get` and `put` etc. unless you know exactly what you are doing! Use `get` and `put` etc. as little as possible.

A function that uses the process dictionary can be rewritten by introducing a new argument.

Example:
Don’t program like this:

```
tokenize([H|T]) ->
    ...;
tokenize([]) ->
    case get_characters_from_device(get(device)) of % Don’t use get/1!
        eof -> [];
        {value, Chars} ->
            tokenize(Chars)
    end.
```

The correct solution:

```
tokenize(_Device, [H|T]) ->
   ...;
tokenize(Device, []) ->
   case get_characters_from_device(Device) of % This is better
      eof -> [];
      {value, Chars} ->
         tokenize(Device, Chars)
   end.
```

The use of `get` and `put` will cause a function to behave differently when called with the same input at different occasions. This makes the code hard to read since it is non-deterministic. Debugging will be more complicated since a function using `get` and `put` is a function of not only of its argument, but also of the process dictionary. Many of the run time errors in Erlang (for example `bad_match`) include the arguments to a function, but never the process dictionary.

6.6 Don’t use import

Don’t use `-import`, using it makes the code harder to read since you cannot directly see in what module a function is defined. Use `exref` (Cross Reference Tool) to find module dependencies.

6.7 Exporting functions

Make a distinction of why a function is exported. A function can be exported for the following reasons (for example):

- It is a user interface to the module.
- It is an interface function for other modules.
- It is called from `apply`, `spawn` etc. but only from within its module.

Use different `-export` groupings and comment them accordingly. 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]).
```

7 Specific Lexical and Stylistic Conventions

7.1 Don’t write deeply nested code

Nested code is code containing `case/if/receive` statements within other `case/if/receive` statements. It is bad programming style to write deeply nested code - the code has a tendency to drift across the page to the right and soon becomes unreadable. Try to limit most of your code to a maximum of two levels of indentation. This can be achieved by dividing the code into shorter functions.

7.2 Don’t write very large modules

A module should not contain more than 400 lines of source code. It is better to have several small modules than one large one.

7.3 Don’t write very long functions

Don’t write functions with more than 15 to 20 lines of code. Split large function into several smaller ones. Don’t solve the problem by writing long lines.

7.4 Don’t write very long lines

Don’t write very long lines. A line should not have more than 80 characters. (It will for example fit into an A4 page.)

In Erlang 4.3 and thereafter string constants will be automatically concatenated. Example:

```
io:format("Name: ~s, Age: ~w, Phone: ~w ~n"
          "Dictionary: ~w.~n", [Name, Age, Phone, Dict])
```

7.5 Variable names

Choose meaningful variable names - this is very difficult.

If a variable name consists of several words, use “_” or a capitalized letter to separate them. Example: `My_variable` or `MyVariable`.

Avoid using “_” as don’t care variable, use variables beginning with “_” instead. Example: `_Name`. If you at a later stage need the value of the variable, you just remove the leading underscore. You will not have problem finding what underscore to replace and the code will be easier to read.

7.6 Function names

The function name must agree exactly with what the function does. It should return the kind of arguments implied by the function name. It should not surprise the reader. Use conventional names for conventional functions (`start`, `stop`, `init`, `main_loop`).

Functions in different modules that solves the same problem should have the same name. Example: `Module:module_info()`.

Bad function names is one of the most common programming errors - good choice of names is very difficult!

Some kind of naming convention is very useful when writing lots of different functions. For example, the name prefix "`is_`" could be used to signify that the function in question returns the atom true or false.

```
is_...() -> true | false
check_...() -> {ok, ...} | {error, ...}
```

7.7 Module names

Erlang has a flat module structure (i.e. there are not modules within modules). Often, however, we might like to simulate the effect of a hierarchical module structure. This can be done with sets of related modules having the same module prefix.

If, for example, an ISDN handler is implemented using five different and related modules. These module should be given names such as:

```
isdn_init
isdn_partb
isdn_...
```

7.8 Format programs in a consistent manner

A consistent programming style will help you, and other people, to understand your code. Different people have different styles concerning indentation, usage of spaces etc.

For example you might like to write tuples with a single comma between the elements:

```
{12,23,45}
```

Other people might use a comma followed by a blank:

```
{12, 23, 45}
```

Once you have adopted style - stick to it.

Within a larger project, the same style should be used in all parts.

8 Documenting Code

8.1 Attribute code

You must always correctly attribute all code in the module header. Say where all ideas contributing to the module came from - if your code was derived from some other code say where you got this code from and who wrote it.

Never steal code - stealing code is taking code from some other module editing it and *forgetting* to say who wrote the original.

Examples of useful attributes are:

```
-revision('Revision: 1.14 ').
-created('Date: 1995/01/01 11:21:11 ').
-created_by('eklas@erlang').
-modified('Date: 1995/01/05 13:04:07 ').
-modified_by('mbj@erlang').
```

8.2 Provide references in the code to the specifications

Provide cross references in the code to any documents relevant to the understanding of the code. For example, if the code implements some communication protocol or hardware interface give an exact reference with document and page number to the documents that were used to write the code.

8.3 Document all the errors

All errors should be listed together with an English description of what they mean in a separate document (See “Error Messages” on page 32.)

By errors we mean errors which have been detected by the system.

At a point in your program where you detect a logical error call the error logger thus:

```
error_logger:error_msg(Format, {Descriptor, Arg1, Arg2, ....})
```

And make sure that the line `{Descriptor, Arg1,...}` is added to the error message documents.

8.4 Document all the principle data structures in messages

Use tagged tuples as the principle data structure when sending messages between different parts of the system.

The record features of Erlang (introduced in Erlang versions 4.3 and thereafter) can be used to ensure cross module consistency of data structures.

An English description of all these data structure should be documented (See “Message Descriptions” on page 32.).

8.5 Comments

Comments should be clear and concise and avoid unnecessary wordiness. Make sure that comments are kept up to date with the code. Check that comments add to the understanding of the code. Comments should be written in English.

Comments about the module shall be without indentation and start with three percent characters (%%%), (See “File Header, description” on page 29.).

Comments about a function shall be without indentation and start with two percent characters (%%), (See “Comment each function” on page 27.).

Comments within Erlang code shall start with one percent character (%). If a line only contains a comment, it shall be indented as Erlang code. This kind of comment shall be placed above the statement it refers to. If the comment can be placed at the same line as the statement, this is preferred.

```
%% 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,
......
```

8.6 Comment each function

The important things to document are:

- The purpose of the function.
- The domain of valid inputs to the function. That is, data structures of the arguments to the functions together with their meaning.
- The domain of the output of the function. That is, all possible data structures of the return value together with their meaning.
- If the function implements a complicated algorithm, describe it.
- The possible causes of failure and exit signals which may be generated by `exit/1`, `throw/1` or any non-obvious run time errors. Note the difference between failure and returning an error.
- Any side effect of the function.

Example:

```
%%--------------------------------------------------------------------
%% 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) ->
   ......
```

8.7 Data structures

The record should be defined together with a plan text description. 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 = []}).
```

8.8 File headers, copyright

Each file of source code must start with copyright information, for example:

8.9 File headers, revision history

Each file of source code must be documented with its revision history which shows who has been working with the files and what they have done to it.

```
%%%----------------------------------------------------------------------
%%% 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
%%%----------------------------------------------------------------------
```

8.10 File Header, description

Each file must start with a short description of the module contained in the file and a brief description of all exported functions.

```
%%%----------------------------------------------------------------------
%%% 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
%%%----------------------------------------------------------------------
```

If you know of any weakness, bugs, badly tested features, make a note of them in a special comment, don't try to hide them. If any part of the module is incomplete, add a special comment. Add comments about anything which will be of help to future maintainers of the module.If the product of which the module you are writing is a success, it may still be changed and improved in ten years time by someone you may never meet.

8.11 Do not comment out old code - remove it

Add a comment in the revision history to that effect. Remember the source code control system will help you!

8.12 Use a source code control system

All non trivial projects must use a source code control system such as RCS, CVS or Clearcase to keep track of all modules.

9 The Most Common Mistakes:

- Writing functions which span many pages (See “Don’t write very long functions” on page 23.).
- Writing functions with deeply nested if’s receive’s, case’s etc (See “Don’t write deeply nested code” on page 23.).
- Writing badly typed functions (See “Use tagged return values” on page 19.).
- Function names which do not reflect what the functions do (See “Function names” on page 24.).
- Variable names which are meaningless (See “Variable names” on page 23.).
- Using processes when they are not needed (See “Assign exactly one parallel process to each true concurrent activity in the system” on page 14.).
- Badly chosen data structures (Bad representations).
- Bad comments or no comments at all (always document arguments and return value).
- Unindented code.
- Using put/get (See “Use the process dictionary with extreme care” on page 20.).
- No control of the message queues (See “Flush unknown messages” on page 16. and See “Time-outs” on page 18.).

10 Required Documents

This section describes some of the system level documents which are necessary for designing and maintaining system programmed using Erlang.

10.1 Module Descriptions

One chapter per module. Contains description of each module, and all exported functions as follows:

- the meaning and data structures of the arguments to the functions
- the meaning and data structure of the return value.
- the purpose of the function
- the possible causes of failure and exit signals which may be generated by explicit calls to `exit/1`.

Format of document to be defined later:

10.2 Message Descriptions

The format of all inter-process messages except those defined inside one module.

Format of document to be defined later:

10.3 Process

Description of all registered servers in the system and their interface and purpose.

Description of the dynamic processes and their interfaces.

Format of document to be defined later:

10.4 Error Messages

Description of error messages

Format of document to be defined later: