Organização de Computadores – 2005/2006
Regras da Programação em Erlang

Paulo Ferreira
paf@dei.isep.ipp.pt

Março de 2006

Conteúdo

- Como escrever código
- Como estruturar o código
- O que se deve fazer
- O que não se deve fazer
- Traduzido/Copiado do documento interno da Ericsson EPK/NP 95:035
- Autores: M. Williams e Joe Armstrong

Princípios gerais

Distinção

- Módulos
- Processos
- Funções

Funções Puras

- Dados os mesmos argumentos, a função devolve o mesmo valor, que é independente do contexto da chamada da função.
- Este é o comportamento normal de uma *função matemática*.
- Uma função que não é pura tem *efeitos laterais*.

Efeitos Laterais

- Envio de uma mensagem
- Recepção de uma mensagem.
- Chama `exit()`
- Chama uma *bif* que muda o ambiente ou o modo de funcionamento
- Exemplos:
  - `get/1, put/2, erase/1, process_flag/2`

Engenharia de Software

- Como fazer software correcto correctamente

Exportação

- Exportar o menor número de funções de um módulo
  - De todas as funções dos módulos só algumas são exportadas,
  - A complexidade de um módulo depende do número de funções, para uma visão externa.
  - O *utilizador* de um módulo precisa de compreender menos funções
  - O *programador* do módulo tem uma maior liberdade para mudar o funcionamento interno do módulo.

Dependências

- Redução de dependências entre módulos
  - Quantos módulos são utilizados de um dado módulo?
  - As dependências formam uma àrvore ou um grafo com ciclos?

Reutilização

- Código usado muitas vezes
  - Deve ir para bibliotecas de funções
  - Biblioteca deve ser um conjunto de funções relacionadas entre si
  - Uma biblioteca de funções para manipulação de listas é uma boa ideia
  - Uma biblioteca para listas e filas é uma má ideia.
  - As melhores funções não possuem efeitos laterais.

≪Limpeza≫

- O código “tricky” ou “dirty” deve ser isolado
  - Se o problema pode ser resolvido usando uma mistura de código *limpo* e *sujo* os dois tipos de código podem ser separados em módulos diferentes
  - Exemplos de código ≪sujo≫:
    - Usar o dicionário do processo.
    - Usar `erlang:process_info/1` para coisas estranhas
    - Fazer coisas que não devia (mas tem de fazer)

Funções

- Quem chama uma função é que sabe o que quer fazer com o resultado!
- Exemplo: quem implementa uma função não deve assumir que quando os argumentos são inválidos, devem ser imprimidos

Mau código

- Não devemos escrever isto:

```
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.
```

Bom código

- Devemos escrever o seguinte:

```
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).
```

Reutilização

- Padrões comuns de código ou comportamento
  - Quando temos o mesmo padrão de código em dois ou mais sítios devemos isolar o código numa função comum em vez de ter o código em dois lugares diferentes.
  - Código *copy-paste* é mais difícil de manter.
  - Se o código não é 100% igual, pode ficar 100% igual.

Top-Down

- Metodologia Top-Down
  - Primeiro: Funcionamemento geral.
  - Depois: Detalhes
  - Vantagens: o código fica independente dos detalhes e da representação dos dados.

Optimização

- Não se deve optimizar o código
  - Primeiro, pôr em funcionamento.
  - Depois, se fôr preciso, optimizar (mantendo o código correcto).
  - Se o código estiver claro, depois é mais fácil de optimizar.

Espanto

- Princípio do *menor espanto*
  - O sistema deve sempre responder de uma forma que cause um *menor espanto* ao utilizador.
  - Um sistema *consistente* onde diferentes módulos se portem da mesma forma é mais fácil de compreender
  - Se ficarmos *espantados* com o que uma função faz, a função resolve o problema errado ou tem o nome errado

Efeitos Laterais

- Eliminação dos *efeitos laterais*
  - Existem várias primitivas com *efeitos laterais*. Se tivermos funções que as usem, estas não pode ser fácilmente reutilizadas, uma vez que causam mudanças permanentes ao seu ambiente.
  - Devemos escrever o máximo de código livre de *efeitos laterais*. Devemos juntar todas funções com efeitos laterais e documentá-las de uma forma correcta.

Modularização

Estruturas de Dados

- As estruturas de dados privadas não devem *transbordar* para fora dos módulos
  Como exemplo, um módulo chamado `queue` para a implementação de filas:

```
-module(queue).
-export([add/2, fetch/1]).
add(Item, Q) ->
  lists:append(Q, [Item]).
 fetch([H|T]) ->
  {ok, H, T};
fetch([]) ->
  empty.
```

Problema

- Esta implementação usa listas para implementar filas mas infelizmente o utilizador tem saber isso.
- Exemplo de utilização:

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

- Problemas:
  - o utilizador precisa de saber que se trata de uma lista
  - o programador não pode mudar a implementação interna

Resolução

- Exemplo:

```
-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.
```

Resultado

- O que podemos fazer agora?

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

- Mas se o utilizador quiser saber o comprimento de um fila pode ser tentado a escrever isto:

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

Correcção

- Nova iteração:

```
-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).
```

Razões

- Porque é que fizemos isto?
  - Temos todos os *detalhes de implementação* escondidos enquanto todas as operações necessárias estão visíveis.
  - Desta forma podemos mudar a implementação sem mudar o código dependente desse módulo.

Melhor

- Módulo queue:

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

Previsibilidade / Erros

Código Determinístico

- Fazer o código o mais determinístico possível
  - Um programa determinístico é um program que corre da mesma maneira, independentemente do número de vezes que o programa correr. Um programa não-determínistico pode dar resultados diferentes de cada vez que correr.
  - Para o *debugging* é uma boa ideia tornar o código o mais determinístico possível. Desta forma os erros podem-se reproduzir de uma forma mais fácil.
  - Por exemplo: um processo tem de arrancar outros cinco processos e depois verificar se eles efectivamente estão a correr, não sendo importante a ordem de arranque dos cinco processos.
  - Podemos arrancar todos os processos e só depois verificar o seu estado, ou arrancar um de cada vez, verificando o seu estado antes de arrancar o processo seguinte.
  - A última opção é a mais indicada.

Defesa

- Não programar *à defesa*
  - Um programa defensivo é um no qual o programdor não confia nos dados de *entrada* do sistema que está a programar.
  - De uma forma geral não se deve testar os argumentos de uma função para ver se estão correctos.
  - A maioria do código de um sistema deve ser escrita assumindo que os dados de entrada estão correctos.
  - Só uma pequena parte deve verificar os dados, quando estem *entram* no sistema pela primeira vez, assumindo-se que a partir daí eles *estão correctos*.

Exemplo


```
%% 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.
```

- Se o argumento `Option` for diferente de `normal` ou `all` a função *estoura*, que é o que deve fazer.
  Quem chama a função é que tem de fornecer os argumentos correctos!

Cuidados

- Isto só é válido para o Erlang!
- Porque sabemos que uma função com argumentos incorrectos *estoura*!
- Noutras linguagens o programa continua para ir *estourar* mais à frente (se tivermos sorte)!
- Em Erlang podemos detectar o *estouro* e corrigir o problema.

Hardware

- Isolar o hardware com um *device driver*
  - Os *device drivers* devem implementar interfaces com o *hardware* que façam o hardware aparecer como um processo Erlang.
  - O hardware deve aparentemente enviar e receber mensagens e comportar-se como um processo quando aparecerem erros.

Fazer e desfazer

- Fazer e desfazer na mesma função
  - Supondo que temos um programa que abre um ficheiro, faz coisas com esse ficheiro, e depois fecha-o. Isso deve ser codificado da seguinte forma:

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

  - Note-se que abrimos (`file:open`) e fechamos o ficheiro (`file:close`) na mesma rotina.

Mau exemplo

- Não devemos programar assim:

```
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
```

Normal/Erros

- Separar o tratamento de erros dos casos *normais*
  - O código dos casos normais não deve ter *à mistura* o código destinado ao tratamento das excepções.
  - O programador deve preocupar-se apenas com o funcionamento normal.
  - Se o código normal falhar, o processo deve comunicar o erro (logs) e *crashar* o mais depressa possível.
  - Não devemos corrigir o erro e continuar.
  - Isso deve ser feito por outro processo.
  - Esta separação simplifica a concepção global do sistema.
  - Os logs gerados quando há erros de hardware ou software devem ser guardados, pois serão usados mais tarde para diagnosticar e corrigir os erros.

Error Kernel

- Identificar o *error kernel*
  - Um dos passos básicos do desenho de um sistema consiste em identificar que partes do sistema têm de estar correctas e que partes do sistema não têm de estar correctas (podem não estar correctas).
  - No desenho convencional de um sistema operativo assume-se que o kernel está correcto enquanto as aplicações dos utilizadores, podem não o estar.
  - Assim se uma aplicação falhar isso deve apenas afectar a aplicação em causa e não o sistema todo.
  - A parte do sistema que tem de estar correcta e assegura o funcionamento de todo o sistema recebe o nome de *error kernel*.
  - O *error kernel* pode possuir uma base de dados *real-time* residente em memória, que guarda o estado do hardware.

Processos

Processos/Módulos

- Um processo por módulo
  - O código para implementar um processo deve estar apenas num módulo.
  - Um processo pode chamar quaisquer funções, mas o *ciclo principal* deve estar apenas num módulo.
  - Ao contrário, devemos ter apenas um tipo de processo por módulo.

Estruturação

- Usar processos para estruturar o sistema
  - Os processos são o elemento básico na construção de um sistema.
  - No entanto não devemos usar processos quando uma chamada a uma função é suficiente.

Registo de processos

- Os processos devem ser ser registados com o mesmo nome do seu módulo. Isto facilita a procura do código correspondente a um processo.
- Só devemos registar processos que devem viver muito tempo.

Quantos processos?

- Atribuir exactamente um processo paralelo a cada actividade realmente concorrente
- Quando devemos decidir entre processos paralelos ou sequenciais, então a estrutura *original* do problema a resolver deve ser usada.
- A regra principal é: – Usar um processo paralelo para modelizar cada actividade realmente concorrente no mundo real.
- Se existir uma correspondência um para um, entre o número de processos paralelos e as actividades paralelas no mundo real, o programa será fácil de compreender.

Papel dos processos

- Cada processo deve ter apenas um *papel*
- Os processos podem ter papeis diferentes no sistema, por exemplo, no modelo cliente-servidor.
- Na medida do possível cada processo deve ter apenas um papel. Por exemplo: um processo pode ser um cliente ou um servidor, mas não deve combinar os dois papeis.
- Outros papeis normais:
  - *Supervisor*: toma conta de outros processos e reinicializa-os em caso de falha..
  - *Worker*: um processo normal (pode ter erros).
  - *Trusted Worker*: não pode ter erros.

Funções genéricas

- Uso de funções genéricas para servidores e protocolos
  - Em muitas circunstâncias é uma boa ideia usar o *generic server* implementado nas bibliotecas do sistema.
  - O mesmo pode ser feito para o processamento de protocolos de comunicação.

Mensagens

Etiquetas

- Mensagens sempre com etiquetas
  - Todas as mensagens devem ser etiquetadas com àtomos.
  - Isto elimina a importância da ordem de recepção das mensagens.
  - Torna mais fácil a implementação de novas mensagens.

Mau exemplo

- Não devemos programar assim:

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

- Se quisermos uma nova mensagem {`get_status_info, From, Option`} esta vai estar em conflito se for colocada antes da mensagem {`Mod, Func, Args`}.

Comunicação síncrona

- Se as mensagens são para comunicação síncrona, a mensagem de retorno deve ser etiquetada com um novo àtomo descrevendo a mensagem devolvida.
- Examplo: se a mensagem recebida tem a etiqueta `get_status_info`, a mensagem de resposta pode ser etiquetada com `status_info`.
- Convém que as etiquetas sejam diferentes para que o *debugging* seja mais fácil.

Exemplo

- Uma boa solução:

```
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.
```

Limpar mensagens

- Deitar fora mensagens desconhecidas
  - Cada servidor deve ter uma alternativa `Other` em pelo menos um dos `receives`.
  - Nota: um servidor pode ter mais do que um `receive`.
  - Isto serve para limpar as filas de espera das mensagens.

Exemplo


```
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.
```

Servidores

Servidores recursivos

- Os servidores devem ser *tail-recursive*
- Se não o forem, o servidor irá alocar memória até que que o sistema fique sem memória disponível.
- Trata-se de um erro subtil.
- Aparentemente tudo funciona, mas ao fim de algum tempo (quanto?)...
- Se usarmos uma biblioteca de *servidores* evitamos este erro.

Má programação


```
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!
```

Programação correcta


```
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
```

Funções de interface

- Deve-se usar funções de interface sempre que possível, e evitar enviar mensagens directamente.
- O protocolo das mensagens é uma informação interna e deve estar *escondido* dos outros módulos.

Exemplo


```
-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>...
```

Time-outs

- Cuidados a ter com o `after` nos `receives`.

  1. Depois de um *time-out*, a mensagem ainda pode chegar.
  2. Será que vai ser processada?
  3. Se não for processada, deve ser *deitada fora*.

Sinais de exit

- Deve haver poucos processos que detectem sinais de exit.
- Os processos devem ter a detecção de sinais activa ou não.
- É uma muito má prática o ligar/desligar da detecção de sinais de exit.
- Nota: deve-se activar a detecção de sinais de exit antes de fazer `spawnlink` de outros processos.

Vários

Uso de records

- O uso de records como estrutura de dados principal é aconselhado.
- Se o record for usado por vários módulos, a sua definição deve ser colocada num ficheiro de definição (com a extensão `.hrl`) que será incluído.
- Para a passagem de dados entre módulos é aconselhado o uso de records.

Selectores e construtores

- O uso de construtores e selectores próprios dos records é aconselhado.
- Não devemos assumir que o record é um tuplo e manipular directamente esse tuplo.
- Exemplo:

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

- Não devemos programar assim:

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

Etiquetas no retorno

- Valores de retorno com etiquetas
- Não devemos programar assim:

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

- Assim o tuplo Key, Value não pode ter o valor `false`.

Corrigindo

- Solução correcta:

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

catch e throw

- Cuidado extremo no uso de `catch` e `throw`
  - Não devemos usar `catch` e `throw` excepto se soubermos exactamente o que estamos a fazer.
  - Devemos usar `catch` e `throw` o menor número de vezes possível.
  - Este mecanismo pode ser útil quando o programa tem de processar dados complicados e poucos fiáveis (do mundo real, não de dentro de um programa) que podem causar erros em muitos lugares dentro do código.
  - Um exemplo desse tipo de programas é um compilador.

Dicionário do processo

- Usar o dicionário do processo com um cuidado extremo
  - Não se deve usar `get` e `put` excepto se soubermos exactamente o que estamos a fazer.
  - Devemos usar `get` e `put` o menor número de vezes possível.
  - Uma função que use o dicionário do processo, pode ser reescrita introduzindo um novo argumento.

import

- Não usar import
  - Não devemos usar `-import`, porque o código fica mais difícil de ler porque deixamos de saber directamente em que módulo é que uma função está definida.
  - Devemos usar o `exref` (Cross Reference Tool) para encontrar dependências entre módulos.

Export

- Exportação de funções
  - Deve-se assinalar porque é que uma função é exportada.
  - Uma função pode ser exportada por várias razões:
    1. Faz parte da interface do *utilizador*.
    2. Vai ser usada/chamada por outros módulos.
    3. Vai ser usada por `apply/spawn` mas apenas dentro do próprio módulo.

Exemplo

- Exportação de funções
  - Devemos agrupar as diferentes funções e comentar os `exports`.

```
%% 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]).
```

Código encaixado

- Não usar muitos níveis de código *encaixado*
  - Código *encaixado* é código que contém muitos `case/if/receive` dentro de outros `case/if/receive`.
  - Neste caso o código tem a tendências de deslizar para a direita na página, e ficar ilegível.
  - Devemos limitar o nosso código a um máximo de dois níveis de indentação.
  - Isto pode ser feito se dividirmos o nosso código em funções mais pequenas.

Tamanhos

- Não escrever módulos muito grandes
  - Um módulo não deve ter mais de 400 linhas de código.
  - É melhor ter vários módulos pequenos do que um módulo grande.
- Não escrever funções muito grandes
  - Não devemos escrever funções com mais do que 15 a 20 linhas de código.
  - Devemos partir as funções mais pequenas.
  - Isto não deve ser resolvido escrevendo linhas maiores.

Tamanho linhas

- Não devemos escrever linhas muito longas
- Uma linha não deve ter mais de 80 caracteres.
- Isto para caber numa página A4 sem problemas.
- No Erlang 4.3 e versões mais recentes a concatenação de strings (constantes) é automática.
- Exemplo:

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

Nomes das variáveis

- Devemos escolher nomes de variáveis com significado – isto é **muito difícil**.

Nomes das variáveis

- Devemos escolher nomes de variáveis com significado – isto é **muito difícil**.
- Se o nome de uma variável possui várias palavras devemos usar o «_» ou uma maiúscula para separar as palavras. Exemplo: `My_variable` ou `MyVariable`.
- Devemos evitar o uso de «_» como variável anónima, usando em sua substituição nomes de variáveis que começem com «_». Exemplo: «`_Nome`».
- Assim, se mais tarde necessitarmos de usar a variável, basta remover o «_» no início do nome.
- Não teremos de procurar qual o «_» que corresponde à variável que queremos utilizar.

Nomes de funções

- O nome da função deve concordar exactamente com o que a função faz. Esta deve usar/devolver o tipo de argumentos que o seu nome deixa entender.
- Devemos usar os nomes *normais* para as funções normais (`start`,`stop`,`init`,`main_loop`).
- Funções em módulos diferentes que resolvem o mesmo problema devem ter o mesmo nome. Exemplo: `Module:module_info()`.

Nomes de funções

- O nome da função deve concordar exactamente com o que a função faz. Esta deve usar/devolver o tipo de argumentos que o seu nome deixa entender.
- Devemos usar os nomes *normais* para as funções normais (`start`,`stop`,`init`,`main_loop`).
- Funções em módulos diferentes que resolvem o mesmo problema devem ter o mesmo nome. Exemplo: `Module:module_info()`.
- Os maus nomes de funções são um dos erros mais comuns de programação - a boa escolha de nomes é muito difícil.
- Devem existir certas convenções para facilitar a escrita de muitas funções diferentes. Por exemplo, o prefixo `is_` pode ser usado para indicar que a função em causa devolve `true` ou `false`.

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

Nomes dos módulos

- O Erlang tem normalmente uma estrutura de módulos plana (não havia módulos dentro de módulos). No entanto podemos querer simular o efeito de uma estrutura de módulos hierárquica.
- Isto pode ser feito com conjuntos de módulos relacionados que possuam o mesmo prefixo no nome.
- Por exemplo, um sistema para a implementação de ISDN foi implementado em vários módulos diferentes. Os módulos deverão ter nomes tais como:
  - `isdn_init`
  - `isdn_core`
  - `isdn_monitor`
  - `isdn_accounting`

Formatação

- A formatação dos programas deve ser consistente
  - Um estilo de programação consistente ajuda o próprio e os outros a compreender o código. Pessoas diferentes possuem diferentes estilos de escrita fazendo uma indentação deferente, um uso diferente dos espaços, etc.
  - Por exemplo, alguns escrevem os tuplos apenas com uma vírgula entre os elementos, enquanto outros preferem uma vírgula seguida por um espaço.
    - `{12,23,45}`
    - `{12, 23, 45}`
  - Depois de adoptarmos um estilo, devemos mantê-lo.
  - Num projecto grande o mesmo estilo deve ser usado em todas as partes.

Documentação

Autoria do código

- Devemos sempre documentar a autoria de todo o código no cabeçalho do módulo.
- Devemos dizer de onde vieram todas as ideias que contribuiram para o módulo - se o código foi derivado de outro código devemos dizer onde o fomos buscar e quem o escreveu.
- Nunca devemos roubar código - roubar código é tirar o código de outro módulo, editar-lo e depois não dizer quem escreveu o original.

Atributos

- Exemplos de atributos úteis:

```
-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').
```

Referências

- Devemos referenciar no código quaisquer documentos que sejam relevantes na compreensão do código.
- Por exemplo, se o código implementa qualquer protocolo de comunicações ou interface de hardware, devemos referenciar o(s) documento(s) e números de página do(s) documento(s) usados na escrita do código.

Documentação dos erros

- Todos os erros devem ser listados, juntamente com uma descrição do seu significado num documento separado.
- Por erros, queremos dizer erros que foram detectados pelo sistema.
- Quando tivermos detectado um erro lógico devemos chamar o `error_logger`: `error_logger:error_msg(Format, {Descriptor, Arg1, Arg2, ....})`
- E devemos ter a certeza que a documentação correspondente à mensagem em causa `{Descriptor, Arg1,...}` está incluída na documentação dos erros.

Estruturas de dados

- Documentar todas as estuturas de dados usadas nas mensagens
  - Devemos usar tuplos etiquetados como a principal estrutura de dados quando enviamos mensagens entre as diferentes partes do sistema.
  - Os records podem ser usados para assegurar a consistência das estruturas de dados entre os diferentes módulos do sistema.
  - Devemos documentar todas as estruturas de dados usadas.

Comentários

Em geral

- Os comentários devem ser claros e concisos, evitando palavreado desnecessário.
- Os comentários devem ser actualizados junto com o código.
- Os comentários devem ajudar a compreensão do código.

Convenções

- Os comentários sobre os módulos devem começar no início das linhas e devem ter no seu início três caracteres de percentagem (%%%).
- Os comentários sobre as funções devem começar no início das linhas e devem ter no seu início dois caracteres de percentagem (%%).
- Os comentários sobre o código devem começar apenas com um caracter de percentagem. O comentário deve estar indentado da mesma forma que o código, na linha anterior ao código, ou ainda melhor na mesma linha do código.

Exemplo


```
%%%  Comment

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

Função

- O que devemos documentar:
  - O objectivo de cada função.
  - Os valores válidos dos argumentos. Que valores/estruturas de dados é que a função aceita como argumentos, e qual o seu significado.
  - Os valores de saída da função. Quais são eles e qual o seu significado.
  - Se a função implementa um algoritmo complicado, devemos descrever esse algoritmo.
  - As causas de falha ou sinais de exit que possam ser gerados por `exit/1`, `throw/1` ou outros erros de run-time. Note-se a diferença entre uma falha (*estouro*) e o devolver um erro.
  - Qualquer efeito lateral da função.

Exemplo

- Comentários de cada função

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

Estruturas de dados

- Os records devem ser definidos juntamente com uma definição do seu significado.

```
%% 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
%%  A {Key, Value} list (default is the empty list).
%%-----------------------------------------
-record(person, {name, age, phone = [], dict = []}).
```

Copyright

- Cada ficheiro deve começar com a informação do copyright:

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

Revisões

- Cada ficheiro deve possuir a informação das revisões a que foi sujeito, e que diz quem fez o quê.

```
%%%-----------------------------------------------
%%% Revision History
%%%-----------------------------------------------
%%% Rev PA1 Date 960230 Author Fred Bloggs (ETXXXXX)
%%% Initial prerelease. 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
%%%-----------------------------------------------
```

Descrição

- Cada ficheiro deve começar com uma curta descrição do módulo contido no ficheiro e uma breve descrição de todas as funções exportadas.

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

Problemas

- Se existirem fraquezas, bugs, coisas mal testadas, isso deve estar assinalado num comentário especial e não deve estar escondido.
- Se alguma parte do módulo estiver incompleta, isso ser estar documentado.
- Devemos documentar tudo o que seja importante para quem no futuro tenha de fazer a manutenção do módulo.

Código antigo

- Código antigo deve ser removido
- Devemos comentar o facto.
- Se o necessitarmos de volta temos o sistema de controle de revisões.

Código antigo

- Código antigo deve ser removido
- Devemos comentar o facto.
- Se o necessitarmos de volta temos o sistema de controle de revisões.
- Todos os projectos que não sejam triviais devem usar um sistema de controle de revisões estilo RCS, CVS, Subversion ou outros para controlarmos a evolução dos ficheiros.

Conclusão

Resumo

- Os erros mais comuns
  - Escrever funções com muitas páginas
  - Funções com muitas estruturas de controle *encaixadas*.
  - Funções sem *etiquetagem* dos valores de retorno.
  - Nomes de variáveis sem significado.
  - Usar processos quando estes não são necessários.
  - Estruturas de dados mal escolhidas.
  - Comentários inexistentes ou maus (pelo menos argumentos e valor de retorno)
  - Código sem indentação.
  - Uso de `put` e `get`.
  - Filas de mensagens sem controle (limpeza e *timeouts*).