Erlang Patterns Matching Business Needs

Torben Hoffmann
CTO, Erlang Solutions
torben.hoffmann@erlang-solutions.com
@LeHoff
Erlang Patterns Matching Business Needs

& Idioms

Torben Hoffmann
CTO, Erlang Solutions
torben.hoffmann@erlang-solutions.com
@LeHoff
Cracks and a Chasm

To cross the chasm you must talk business value!
Enterprise Software
What Killed the Death Star?
What Killed the Death Star?

PowerShieldBreakdownException
What Killed the Death Star?

PowerShieldBreakdownException

SurfaceWithAlleysDesignException
What Killed the Death Star?

PowerShieldBreakdownException

SurfaceWithAlleysDesignException

MissileEnteredUnprotectedVentilationShaftException
Expect resistance…

Source: http://2.bp.blogspot.com/-qNM3LGTtUYM/UIFLJGd_MLI/AAAAAAAAAnU/GCtI5SYfbCs/s320/orc-army.jpg

Source: http://images1.wikia.nocookie.net/__cb20110119125642/villains/images/e/ef/Saruman.jpg

Source: http://asset3.cbsistatic.com/cnwk.1d/i/tim2/2013/08/12/Larry_Ellison_Oracle_Open_World_2009_610x407.jpg
Expect resistance…

Source: http://2.bp.blogspot.com/-qNM3LGTtUYM/UIFLJGd_MLI/AAAAAAAAnU/GCtl5SYfbCs/s320/orc-army.jpg

Source: http://images1.wikia.nocookie.net/__cb20110119125642/villains/images/e/ef/Saruman.jpg

Source: http://asset3.cbsistatic.com/cnwk.1d/i/tim2/2013/08/12/Larry_Ellison_Oracle_Open_World_2009_610x407.jpg
Expect resistance…

Source: [http://2.bp.blogspot.com/-qNM3LGTtUYM/UIFLJGd_MLI/AAAAAAAAAnI/AAAI5SYfbsM/s320/orc-army.jpg](http://2.bp.blogspot.com/-qNM3LGTtUYM/UIFLJGd_MLI/AAAAAAAAAnI/AAAI5SYfbsM/s320/orc-army.jpg)


Source: [http://images1.wikia.nocookie.net/__cb20110119125642/villains/images/e/ef/Saruman.jpg](http://images1.wikia.nocookie.net/__cb20110119125642/villains/images/e/ef/Saruman.jpg)

Citius, Altius, Fortius
Olympic Motto

Citius, Altius, Fortius
Citius, Maior, Vilius
Business Imperative

Citius, Maior, Vilius
Everlasting Software Requirements
Everlasting Software Requirements

speed to market
Everlasting Software Requirements

speed to market

reliable
Everlasting Software Requirements

speed to market

reliable

scalable
Everlasting Software Requirements

speed to market
reliable
scalable
maintainable
Agile Manifesto
Agile Manifesto

Individuals and interactions > Processes and tools
Agile Manifesto

Individuals and interactions > Processes and tools

Working software > Comprehensive documentation
Agile Manifesto

Individuals and interactions > Processes and tools
Working software > Comprehensive documentation
Customer collaboration > Contract negotiation
Agile Manifesto

Individuals and interactions > Processes and tools
Working software > Comprehensive documentation
Customer collaboration > Contract negotiation
Responding to change > Following a plan
Software Architecture

Separation of concerns
Quality-driven
Recurring styles
Conceptual integrity
Erlang History
There are two ways of constructing a software design:
There are two ways of constructing a software design:
One way is to make it so simple that there are *obviously* no deficiencies…
There are two ways of constructing a software design:
One way is to make it so simple that there are *obviously* no deficiencies…
… and the other way is to make it so complicated that there are no *obvious* deficiencies.
There are two ways of constructing a software design:
One way is to make it so simple that there are *obviously* no deficiencies…
… and the other way is to make it so complicated that there are no *obvious*
deficiencies.

- C.A.R. Hoare
Erlang’s Original Requirements
Erlang’s Original Requirements

Large scale concurrency
Erlang’s Original Requirements

Large scale concurrency
Soft real-time
Erlang’s Original Requirements

Large scale concurrency
Soft real-time
Distributed systems
Erlang’s Original Requirements

- Large scale concurrency
- Soft real-time
- Distributed systems
- Hardware interaction
Erlang’s Original Requirements

Large scale concurrency
Soft real-time
Distributed systems
Hardware interaction
Very large software systems
Erlang’s Original Requirements

- Large scale concurrency
- Soft real-time
- Distributed systems
- Hardware interaction
- Very large software systems
- Complex functionality
Erlang’s Original Requirements

Large scale concurrency
Soft real-time
Distributed systems
Hardware interaction
Very large software systems
Complex functionality
Continuous operation for many years
Erlang’s Original Requirements

Large scale concurrency
Soft real-time
Distributed systems
Hardware interaction
Very large software systems
Complex functionality
Continuous operation for many years
Software maintenance on-the-fly
Erlang’s Original Requirements

Large scale concurrency
Soft real-time
Distributed systems
Hardware interaction
Very large software systems
Complex functionality
Continuous operation for many years
Software maintenance on-the-fly
High quality and reliability
Erlang’s Original Requirements

Large scale concurrency
Soft real-time
Distributed systems
Hardware interaction
Very large software systems
Complex functionality
Continuous operation for many years
Software maintenance on-the-fly
High quality and reliability
Fault tolerance

wanted
wanted productivity
wanted

productivity

no down-time
wanted

productivity

no down-time

something that always works
wanted
wanted

money
wanted

money

money
wanted

money

money

money
wanted

money

money

money

it's a rich man's world!
wanted

money

money

money

it’s a rich man’s world!
If our basic tool, the language in which we design and code our programs, is also complicated, the language itself becomes part of the problem rather than part of its solution.

- C.A.R. Hoare
Good Erlang Domains
Good Erlang Domains

Low latency over throughput
Good Erlang Domains

Low latency over throughput

Stateful (in contrast to being stateless)
Good Erlang Domains

Low latency over throughput
Stateful (in contrast to being stateless)
Massively concurrent
Good Erlang Domains

Low latency over throughput
Stateful (in contrast to being stateless)
Massively concurrent
Distributed
Good Erlang Domains

Low latency over throughput
Stateful (in contrast to being stateless)
Massively concurrent
Distributed
Fault tolerant
Good Erlang Domains

Low latency over throughput
Stateful (in contrast to being stateless)
Massively concurrent
Distributed
Fault tolerant
Uses OTP
Good Erlang Domains

- Low latency over throughput
- Stateful (in contrast to being stateless)
- Massively concurrent
- Distributed
- Fault tolerant
- Uses OTP
- Non-stop operation
Good Erlang Domains

Low latency over throughput
Stateful (in contrast to being stateless)
Massively concurrent
Distributed
Fault tolerant
Uses OTP
Non-stop operation

Under load, Erlang programs usually performs as well as programs in other languages, often way better.

Jesper Louis Andersen
The glove fits!

<table>
<thead>
<tr>
<th>Low latency</th>
<th>Stateful</th>
<th>Massively concurrent</th>
<th>Distributed</th>
<th>Fault tolerant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messaging</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Webservers</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Soft switches</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Distributed DBs</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Queueing systems</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
The Golden Trinity Of Erlang

- fail fast
- share nothing
- failure handling
To Share Or Not To Share
To Share Or Not To Share

Memory
To Share Or Not To Share

Memory

Pl
To Share Or Not To Share

Memory

P1  P2
To Share Or Not To Share

Corrupt

P2
To Share Or Not To Share

Corrupt
To Share Or Not To Share

Corrupt

Memory
To Share Or Not To Share

Corrupt

Memory

PI
To Share Or Not To Share

Corrupt

Memory

Memory

PI
To Share Or Not To Share

Corrupt

Memory

P1

Memory

P2
To Share Or Not To Share

Corrupt

Corrupt

Memory

P2
To Share Or Not To Share

Corrupt

Memory

P2
Failures

Anything that can go wrong, will go wrong

*Murphy*
Failures

Programming errors

Anything that can go wrong, will go wrong

Murphy
Failures

- Anything that can go wrong, will go wrong
- Programming errors
- Disk failures

Murphy
Failures

Anything that can go wrong, will go wrong

Murphy

Programming errors
Disk failures
Network failures
Failures

- Programming errors
- Disk failures
- Network failures

Anything that can go wrong, will go wrong

Most programming paradigmes are fault in-tolerant

Murphy
Failures

Anything that can go wrong, will go wrong

Programming errors
Disk failures
Network failures

Most programming paradigmes are fault in-tolerant
⇒ must deal with all errors or die
Failures

- Programming errors
- Disk failures
- Network failures

Most programming paradigmes are fault in-tolerant
⇒ must deal with all errors or die

Anything that can go wrong, will go wrong

Murphy
Failures

Programming errors
Disk failures
Network failures

Anything that can go wrong, will go wrong

Murphy

Most programming paradigmes are fault in-tolerant
⇒ must deal with all errors or die

Erlang is fault tolerant by design
Failures

Programming errors
Disk failures
Network failures

Anything that can go wrong, will go wrong

Murphy

Most programming paradigms are fault in-tolerant
⇒ must deal with all errors or die

Erlang is fault tolerant by design
⇒ failures are embraced and managed
Failures

Programming errors
Disk failures
Network failures

Anything that can go wrong, will go wrong

Murphy

Most programming paradigmes are fault in-tolerant
⇒ must deal with all errors or die

Erlang is fault tolerant by design
⇒ failures are embraced and managed
Let It Fail

convert(monday) -> 1;
convert(tuesday)  -> 2;
convert(wednesday) -> 3;
convert(thursday) -> 4;
convert(friday)  -> 5;
convert(saturday) -> 6;
convert(sunday)  -> 7;
convert(_) ->
    {error, unknown_day}.
Let It Fail

convert( monday )  ->  1;
convert( tuesday )  ->  2;
convert( wednesday )  ->  3;
convert( thursday )  ->  4;
convert( friday )  ->  5;
convert( saturday )  ->  6;
convert( sunday )  ->  7.
Let It Fail

convert(monday)    -> 1;
convert(tuesday)    -> 2;
convert(wednesday)  -> 3;
convert(thursday)   -> 4;
convert(friday)     -> 5;
convert(saturday)   -> 6;
convert(sunday)     -> 7.

Erlang encourages offensive programming
Intentional Programming

a style of programming where the reader of a program can easily see what the programmer intended by their code. [1]

data retrieval - `dict:fetch(Key, Dict) = Val | EXIT`

the programmer knows a specific key should be in the dictionary and it is an error if it is not.

search - `dict:find(Key, Dict) = {ok, Val} | error.`

it is unknown if the key is there or not and both cases must be dealt with.

test - `dict:is_key(Key, Dict) = Boolean`

knowing if a key is present is enough.
Benefits of let-it-fail

Data Mobility component breakdown

Source: http://www.slideshare.net/JanHenryNystrom/productivity-gains-in-erlang
Benefits of let-it-fail

Source: http://www.slideshare.net/JanHenryNystrom/productivity-gains-in-erlang

code that solves the problem
Benefits of let-it-fail

Source: http://www.slideshare.net/JanHenryNystrom/productivity-gains-in-erlang

code that solves the problem
Benefits of let-it-fail

Data Mobility component breakdown

Source: http://www.slideshare.net/JanHenryNystrom/productivity-gains-in-erlang
Benefits of let-it-fail

Data Mobility component breakdown

Source: http://www.slideshare.net/JanHenryNystrom/productivity-gains-in-erlang

code that solves the problem
Benefits of let-it-fail

Data Mobility component breakdown

Source: http://www.slideshare.net/JanHenryNystrom/productivity-gains-in-erlang
Benefits of let-it-fail

Data Mobility component breakdown

Source: http://www.slideshare.net/JanHenryNystrom/productivity-gains-in-erlang
Benefits of let-it-fail

Data Mobility component breakdown

Source: http://www.slideshare.net/JanHenryNystrom/productivity-gains-in-erlang
Show me the money!

*Function Point Analysis* of the size of the problem

Conservative estimation of the number of inputs, outputs and internal storage

Includes design, box test, system test, project management efforts
Intermezzo
MY SYSTEM
21st Century Edition

by Aron Nimzowitsch

THE LANDMARK POSITIONAL CHESS TRAINING CLASSIC IN AN EASY-TO-STUDY ALGEBRAIC FORMAT / 419 DIAGRAMS

"A thorough knowledge of the elements takes us more than half the road to mastership."
-Aron Nimzowitsch

Edited by Lou Hays

Introduction by International Grandmaster Yasser Seirawan

Copyrighted Material
Language and Models

How many trains on one piece of track?
Language and Models

How many trains on one piece of track?

0..1
Language and Models

How many trains on one piece of track?

0..1 0..N
Language and Models

How many trains on one piece of track?

0..N
Language and Models

How many trains on one piece of track?

0..N

Without a language for something you cannot talk about it!
Visual Erlang
Visual Erlang Objectives
Visual Erlang Objectives

Detailed enough to capture important aspects
Visual Erlang Objectives

Detailed enough to capture important aspects
Not suited for 100% explanation of Erlang
Visual Erlang Objectives

Detailed enough to capture important aspects
Not suited for 100% explanation of Erlang
Standardise on how we show Erlang architecture
Visual Erlang Objectives

Detailed enough to capture important aspects
Not suited for 100% explanation of Erlang
Standardise on how we show Erlang architecture

https://github.com/esl/visual_erlang
Processes in Visual Erlang

- $P \rightarrow O \rightarrow Q$: $P$ monitors $Q$
- $P \rightarrow Q$: $P$ and $Q$ are linked
- $P \rightarrow Q$: $P$ spawns $Q$
- $P \rightarrow M \rightarrow Q$: $P$ sends $M$ to $Q$
- $P \rightarrow \ldots \rightarrow Q$: Exit signal
Functions and Statedata

Function Name
w/ entity N

state data for
a function

Process P has
public API
SomeFun/N &
State data
StateData
Visual Erlang Patterns

Adds vocabulary about architecture

Share insights

Consider failures while designing
Tuple Space Storage Pattern
Manager/Worker Pattern
Supervisor Pattern
Why Document Erlang Patterns?

The knowledge funnel

Concept from R. Martin “The Design of Business”

source: http://christianaaddison.wordpress.com/2011/04/19/week-four-ux-boot-camp-co-design/
Realities of software development

Realities of software development

Business benefits of supervisors
Business benefits of supervisors

Only one process dies
Business benefits of supervisors

Only one process dies

isolation gives continuous service
Business benefits of supervisors

Only one process dies
  isolation gives continuous service

Everything is logged
Business benefits of supervisors

Only one process dies

  isolation gives continuous service

Everything is logged

  you know what is wrong
Business benefits of supervisors

Only one process dies
  isolation gives continuous service
Everything is logged
  you know what is wrong
Corner cases can be fixed at leisure
Business benefits of supervisors

Only one process dies
  isolation gives continuous service
Everything is logged
  you know what is wrong
Corner cases can be fixed at leisure
  Product owner in charge!
Business benefits of supervisors

Only one process dies
  isolation gives continuous service
Everything is logged
  you know what is wrong

Corner cases can be fixed at leisure

Product owner in charge!
Not the software!
Business benefits of supervisors

Only one process dies

- isolation gives continuous service

Everything is logged

- you know what is wrong

Corner cases can be fixed at leisure

Product owner in charge!

Not the software!
Cruising with Erlang
Cruising with Erlang

Understand the failure model
Cruising with Erlang

Understand the failure model

*Embrace failure!*
Cruising with Erlang

Understand the failure model

*Embrace failure!*

Use patterns to deliver business value
Cruising with Erlang

Understand the failure model

*Embrace failure!*

Use patterns to deliver business value

*Stay in charge!*
Cruising with Erlang

Understand the failure model

*Embrace failure!*

Use patterns to deliver business value

*Stay in charge!*