Monkey, Take the Wheel

The cognitive complexity of your projects

Your Speaker

Dmytro Lytovchenko

Sr. Developer and Technical Lead @ Erlang Solutions, Sweden

25 years XP of looking busy at the keyboard

@kvakvs

You may have seen my

- BEAM Wisdoms website
- BEAM VM experiment in Rust

As a developer I'd like to...

- Have less friction while developing, reading, trying to memorize or understand the code
- Do less thinking while maintaining the high quality of my work
- Trivialize some larger code changes

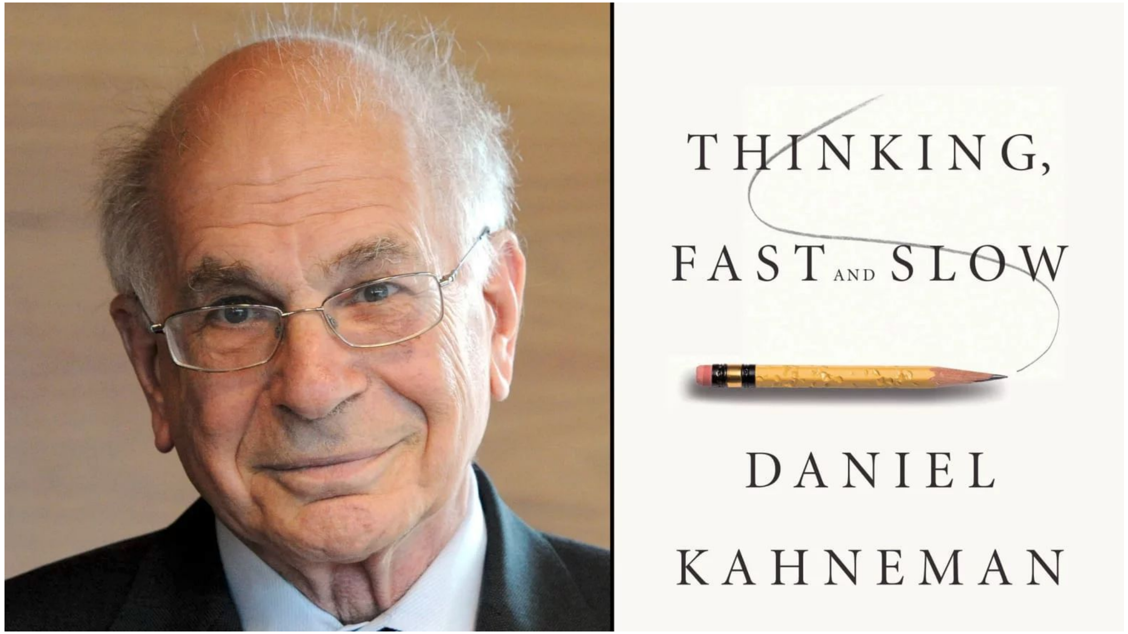
Motivation

To give understanding

- why various “best practices” exist
- what they give you as a developer.

Of Human Brain

“Thinking Fast and Slow”



The human brain has two modes of operation

System 1

System 2

System 1

Fast, automatic, frequent, emotional, stereotypic, unconscious.

- Native language
- Muscle memory: walking, cycling, etc
- Memorized reactions and answers
- Quick judgements

System 1 (dev)

- Relaxed coding, easy algorithms
- Following checklists with simple steps
- Able to understand simple constructs and ideas

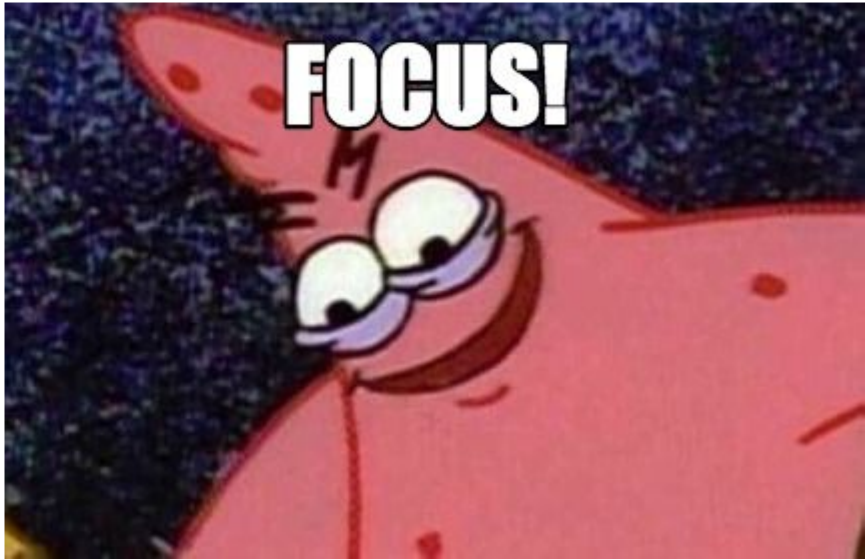
System 2

Slow, effortful, infrequent, logical, calculating, conscious.

- Computations, judgements
- Careful operations, comparisons
- Planning
- Precision work (parking)

System 2 (dev)

- Smart, knowledgeable
- Planning & Design
- Learning new code and concepts
- Code reviews
- Investigations



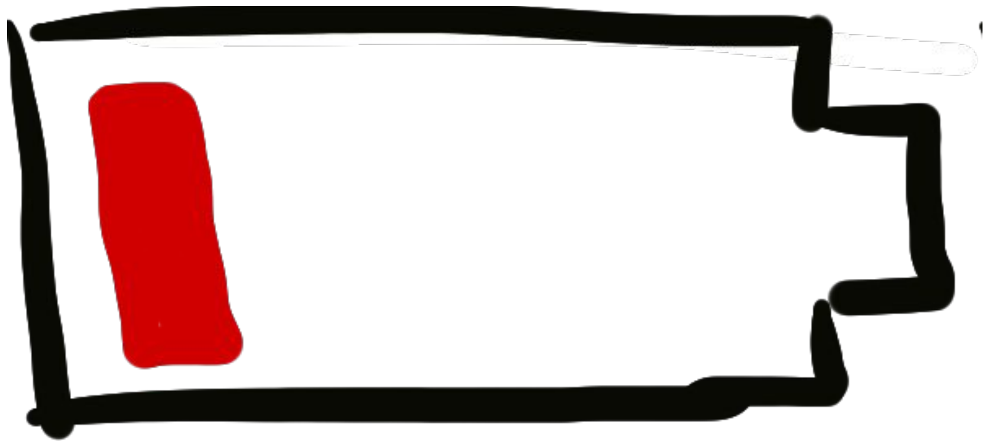
Tim Urban

“Why Procrastinators Procrastinate” (2013)

An article at waitbutwhy.com

TED talk, Feb 2016

Not Sustainable



Organize Your Work (1)

Use System 2 (the smart thinker) for:

- Planning your day
- Designing
- Investigating problems
- Reviewing code

Organize Your Work (2)

Use System 1 (the monkey) for:

- Following the plans
- Creating simple code
- Modifying code and experimenting

LOSING CONTROL
MONKEY, TAKE THE WHEEL
BUT... I CAN'T DRIVE
...

Of Code

Genesis

All code is created equally ~~perfect~~ good.

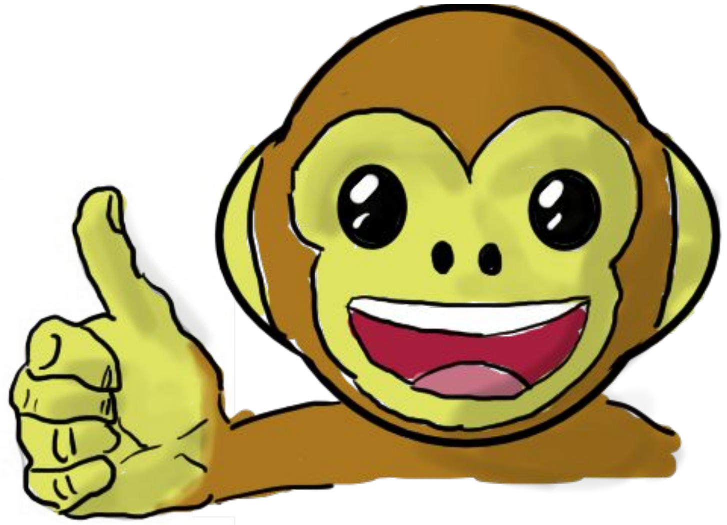
Until the requirements change.

Least Surprise Principle

- Make your project do the expected thing
- Store your code at the expected locations

Tested on primates

Animal-safe and friendly



Assumptions About the Future Developer

- Assume the user has an editor with
  - Code navigation
  - Search
- Assume the user will be happy to
  - run your checks and tests (make sure they know how)

New in the Project

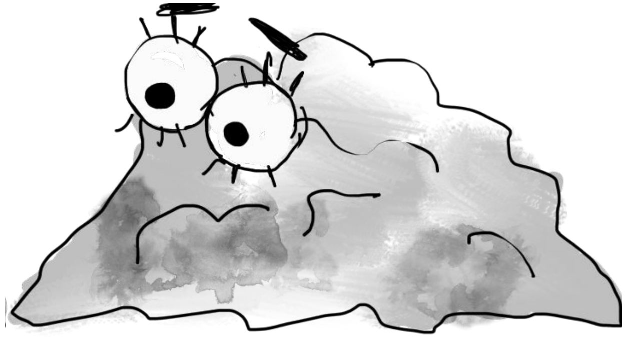
- A lot to learn
- Assume the user has no clue how to
    - Get dependencies
    - Build, test, run
    - Prerequisites: Special directories, files, databases, networks which must exist...

Consider a Better Build System

- Clean build system is important, but hard to do
  - Gather requirements
  - Evolution over revolution
  - Reduce the scope

File & Module Structuring

- Smaller modules
    - Split and regroup your code
    - Module name helps grouping the code
- Elixir: Namespaces are great, use them!


misc_util

Naming Language

- Flow like natural language
  - Functions: start with a verb
  - Predicate functions and boolean variables start with a question:
    - Is? Can? Does? Whether?...
  - Structs/records: form a noun

Visual Structuring (1)

- Why?
  - Ability to clearly see the code structure
  - Reduce visual complexity
  - Shorter time to understand

Visual Structuring (2)

- Aligning assignments
- Aligning struct fields
- Aligning data

```
nil = List.last([])
lower..upper = 1..10
```

```
nil          = List.last([])
lower..upper = 1..10
```

Visual Structuring (3)

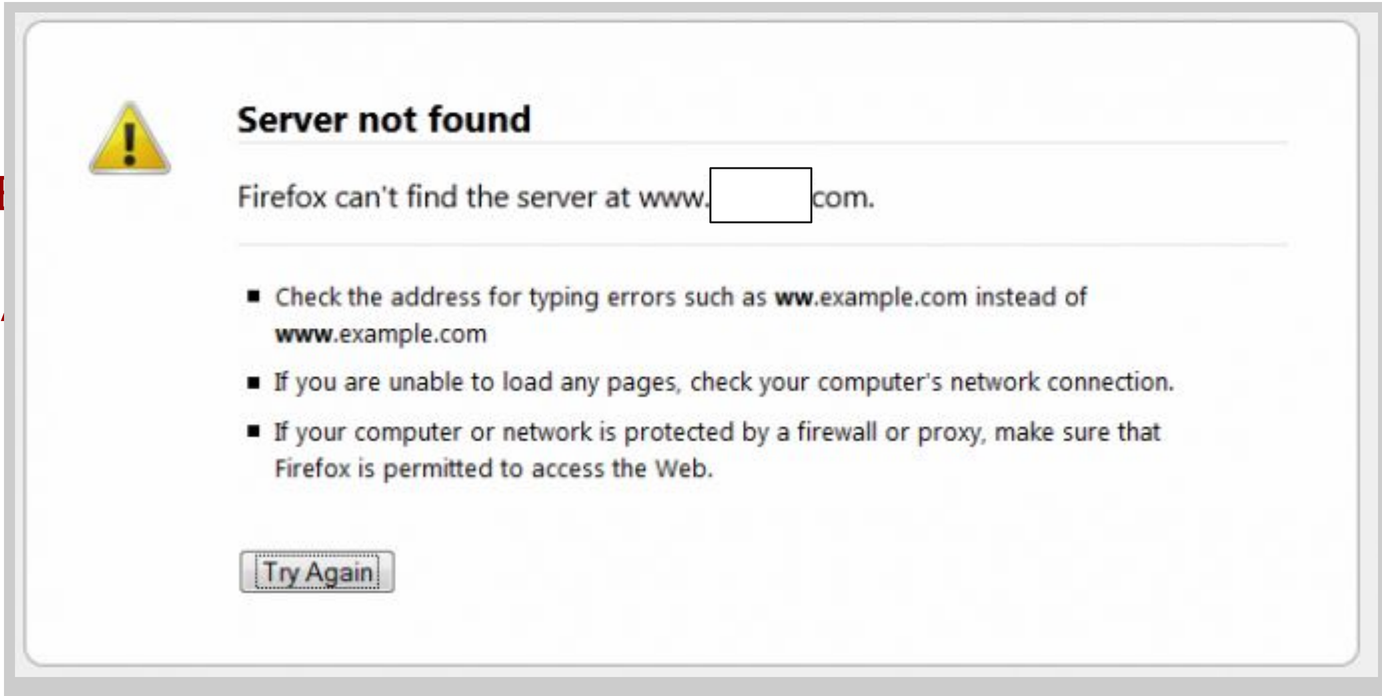
- Short concise functions with comment
  - What it does, why?
  - How to use?

```
%% @doc Spawn a grumble and store
%% into a flexible box
grumble(X) -> box:store(spawn(X)).
```

Visual Structuring (4)

- Documentation too far from code = obsolete

```
%% See READMI
%% PROJ/apps/
```



Server not found

Firefox can't find the server at www.[redacted]com.

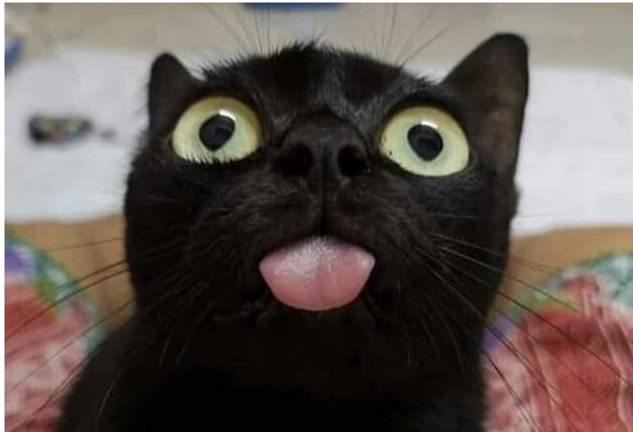
- Check the address for typing errors such as **ww**.example.com instead of **www**.example.com
- If you are unable to load any pages, check your computer's network connection.
- If your computer or network is protected by a firewall or proxy, make sure that Firefox is permitted to access the Web.

Visual Structuring (5)

- Refer to other functions and modules in comments to add context
  - Makes sense only if referenced once

```
%% @doc Used only from cat_app.erl

cat(blep) -> mlem.
```



Cyclomatic complexity

- Code metric
- Number of linearly independent code paths

Reduce code nesting where you can

Break out simpler functions

The metric was developed by Thomas J. McCabe, Sr. in 1976.

Predictable code behaviour

- It does what I think it should do
- Surprises can cost days or weeks of developer time

Predictable Code Placement

- Predictable and consistent naming
- Place things in your code, where they will be found
- Related functions, types, structs together
- Definitions on top

Expected =:= Actual

- Subdirectories and apps, don’t be afraid to move
- Clean and visual boot up sequence for your system
- Remember this code will be read later, by you also

Your Tools

Compile time checks

- Static typing (records, structs)
- Strongly keyed structures, ideally also typed
- Prefer named constants over literals

```
my_long_special_vaue

vs.

?MY_LONG_SPECIAL_VALUE
```

```
:my_long_special_vaue

vs.

const my_long_value, do: …
defmacro my_long_value…
```

Functions with Many Args

- Erlang maps and records
- Elixir records and keywords

```
myfunc(A, B, Time, Count, State, Status,
       Value1, Value2, KeyFrom, KeyTo, Sort, Reverse) ->
Vs.
myfunc(#{a => A, b => B, time => Time, count => Count, …)

Elixir:
def myfunc(a: a, b: b, time: time, count: count, …)
```

Strong typing

- Elixir has structs
    - #Rabbit{legs: 4}
- Erlang has records
    - #rabbit{legs = 4}
- Tagged tuples
    - {rabbit, 4}

Compile-Time Checks

- Matching data on an expected format
  - as close to static typing as you can get

Be loud about bad argument calls!

```
fahrenheit({celsius, C}) -> … ;
fahrenheit({fahrenheit, F}) -> F;
fahrenheit(Otherwise) ->
  erlang:error({badarg, ?MODULE, ?FUNCTION_NAME}).

def fahrenheit(#Temp{t: :celsius, v: c}), do: …
def fahrenheit(#Temp{t: :fahrenheit, v: f}), do: f
def fahrenheit(other), do: raise RuntimeError
```

Static Checking Tools

- Type specs (Dialyzer)
- Static analysis (inaka/elvis)
    - Code smells
    - Code style checks, formatting
        - erl_tidy, erl_prettypr
        - mix format

Runtime checks

- Function guards: Allow enforcing some incoming data types on arguments.
- Safety check macros
    - Make your debug builds more vocal about suspicious things
- Assertions

Mark Your Errors

- Explicit is better than implicit
- Stacktrace sometimes is not available or off by miles!
- Mark where your errors originate from
    - badarg
      vs
      {error, {badarg, ?MODULE, ?FUNCTION_NAME}}

Mark Your Logs

- Always log location where the error was created
- Precise time helps
    - Can match multiple logs

Mark Your Data

- Mark where your data originates from
- Named and explicit is better than implicit

  -
    ```
    {tcp, []}
    vs
    #transport{type = tcp,
               buffer = [],
               created_at = {mymodule, myfunction}}
    ```

Hard to Leave Unfinished

- When doing a large scale change, leave traces everywhere
  - Logs
  - Artificial compile or runtime errors
  - Easy to find comments, e.g.:
    - Use UNFINISHED or TODO in comments
    - Teach your CI or local git hook to fail at “UNFINISHED” or “TODO”

Your Workflow

Choosing the Workflow

Alternating between System 2 and System 1

- Plan your day
- Follow the checklist
- Rest

Choosing the Workflow (2)

- Simple is better
- Shorter is better
- Document the intent
- Prefer automatic checking

Choosing the Workflow (3)

- Minimize distractions and unnecessary manual actions
- Perfectly one click flow
  - “build”, “build+test”, “build+test+release” etc

Thank you!

Dmytro Lytovchenko
@kvakvs