Fighting Complexity

Software has almost no constraints.

Software has almost no constraints.

We can make pigs fly, if we want to.

If you can imagine a system, you can probably build it in software.

The greatest limitation in software is our ability to understand the systems we are creating.

— *John Ousterhout, A Philosophy of Software Design*

As we add new features, our code evolves and naturally becomes more complicated.

Subtle dependencies emerge.

Over time, this complexity accumulates.

Over time, this complexity accumulates.

It becomes harder and harder for us to keep all of the relevant information in our minds …

Because the system is becoming complex and we’re struggling to keep all the details in our minds:

Because the system is becoming complex and we're struggling to keep all the details in our minds:

It becomes harder to add new features … it slows us down.

Because the system is becoming complex and we're struggling to keep all the details in our minds:

It becomes harder to add new features … it slows us down.

We can easily miss some detail and create bugs …
which further slows us down.

The effect multiplies and accelerates with every new contributor to our codebase.

Progress starts to stagnate and costs start to grow rapidly.

Its all about:

Fighting Complexity

Its about building:

Stamina

Simpler, less complex, designs allow us to build larger more powerful systems before complexity becomes overwhelming.

Complexity

Anything related to the structure of a codebase that makes it hard to understand or improve that codebase.

Symptom 1:

Change Amplification

A seemingly simple change requires code modification in many places.

Symptom 2:

Cognitive Load

How much information does a developer need to make a change to the system.

Symptom 3:

Unknown unknowns

It’s not obvious which pieces of code must be modified to complete a task, or what information a developer must have to carry out the task successfully.

Complexity is caused by two things
dependencies *and* **obscurity**.

Tactical Programming

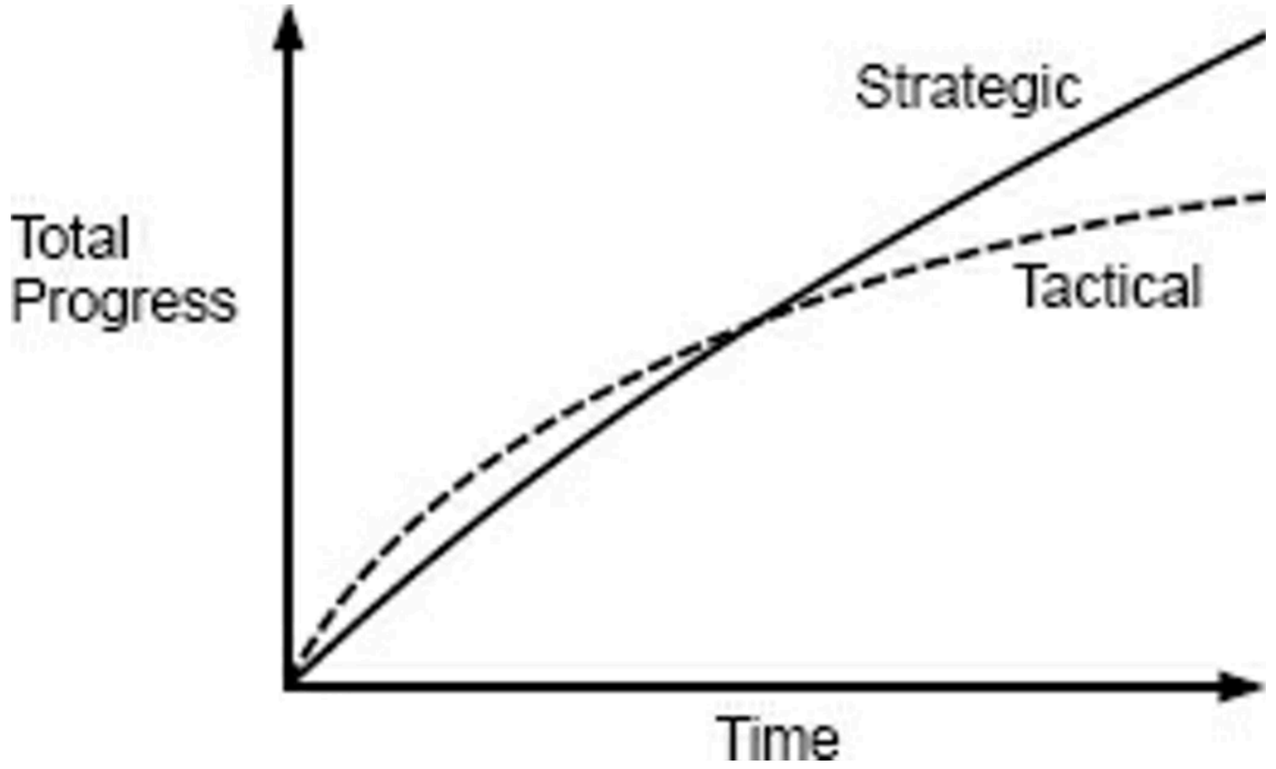
In the tactical approach, our main focus is to get something working. This is how systems become complicated - one small compromise at a time.

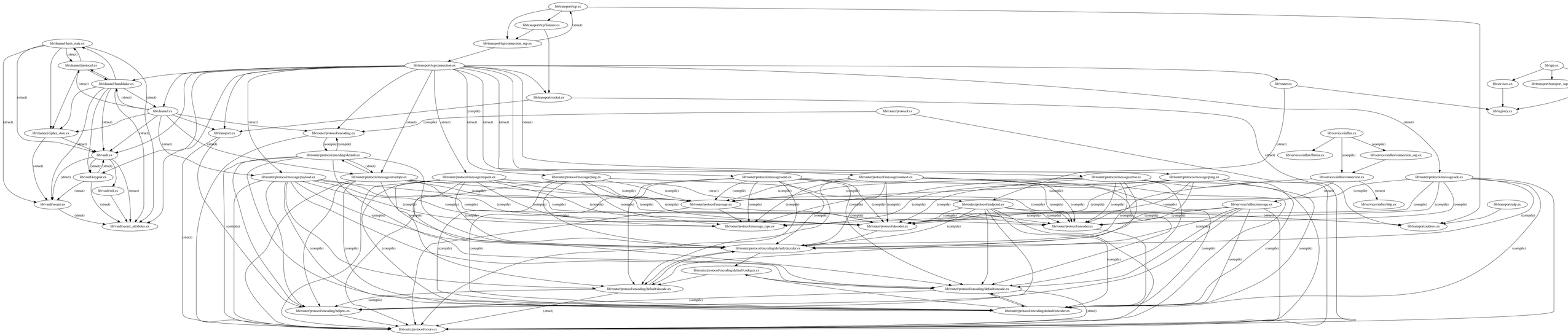
Strategic Programming

The first step towards becoming a good software designer is to realize that working code isn't enough. It's not acceptable to introduce unnecessary complexities in order to finish your current task faster.

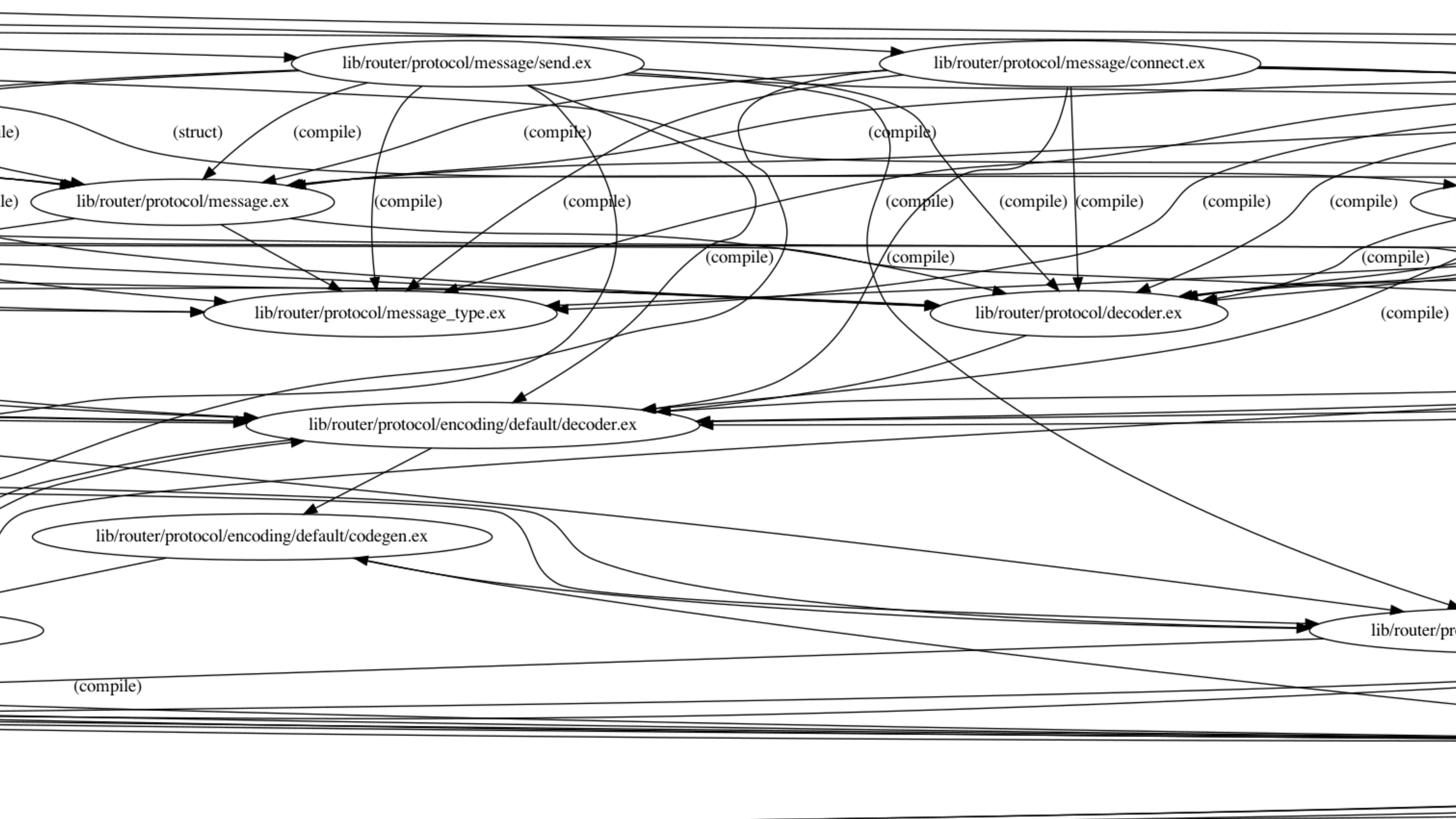
The most important thing is the long-term structure of the system. Our primary goal must be to produce a great design, which also happens to work. This is strategic programming.

– *John Ousterhout. A Philosophy of Software Design*


Strategic
Total
Progress
Tactical
Time



The Tactical Tornado


lib/router/protocol/message/send.ex
lib/router/protocol/message/connect.ex
lib/router/protocol/message.ex
lib/router/protocol/message_type.ex
lib/router/protocol/decoder.ex
lib/router/protocol/encoding/default/decoder.ex
lib/router/protocol/encoding/default/codegen.ex
(struct)
(compile)

The simple parts of reducing Cognitive Load ...

Good names, good comments, consistency, mix format, credo etc.

Modular Design

A developers only need to face a small fraction of the overall complexity at any given time.

Does not mean more modules.

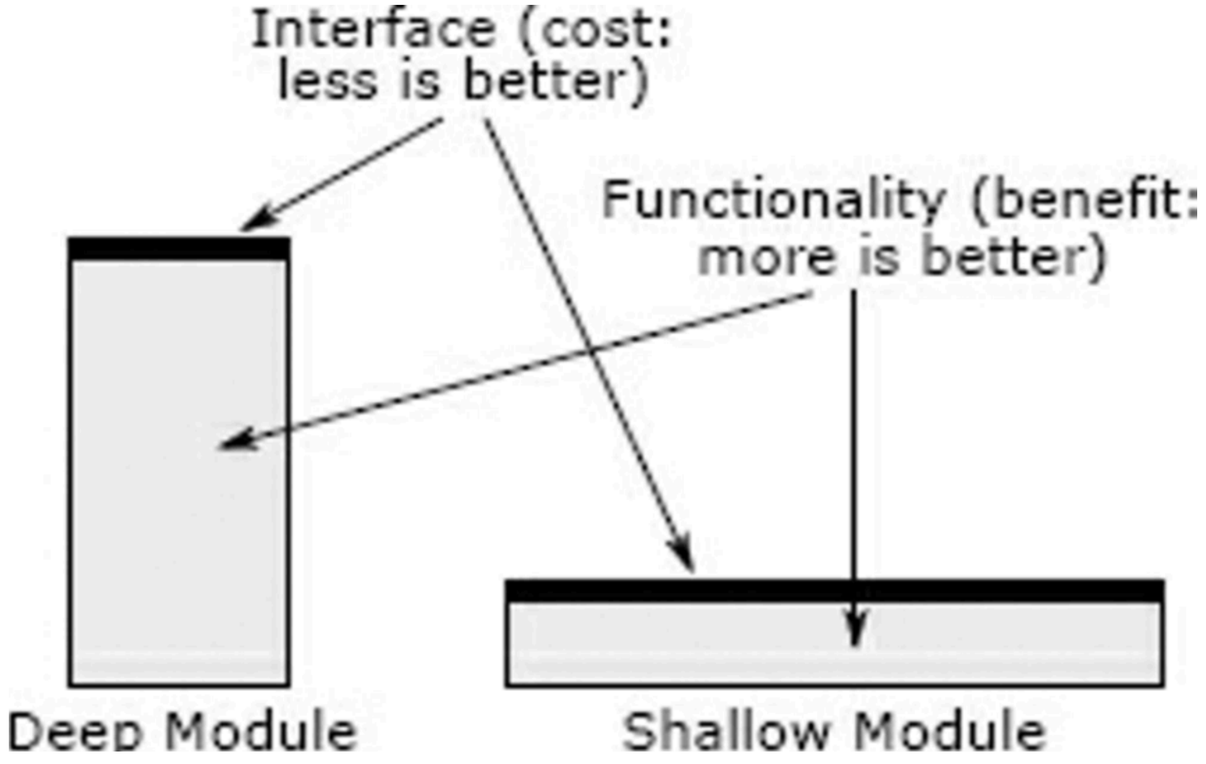
Encapsulation

Encapsulate complexity so our team can work on other parts of the system without being exposed to the complexity encapsulated in this module.

In Elixir we can encapsulate at the function level and at the module level.

Deep Modules

The best modules are those that provide powerful functionality yet have simple interfaces.


Interface (cost:
less is better)
Functionality (benefit:
more is better)
Deep Module
Shallow Module

```
1
2  {:ok, c} = Ockam.SecureChannel.create(vault: v, identity_keypair: k, onward_route: r)
3
```

Flow 

`Flow` allows developers to express computations on collections, similar to the `Enum` and `Stream` modules, although computations will be executed in parallel using multiple `GenStage`s.

Here is a quick example on how to count words in a document in parallel with Flow:

```
File.stream!("path/to/some/file")
|> Flow.from_enumerable()
|> Flow.flat_map(&String.split(&1, " "))
|> Flow.partition()
|> Flow.reduce(fn -> %{} end, fn word, acc ->
  Map.update(acc, word, 1, & &1 + 1)
end)
|> Enum.to_list()
```

Hyrum's Law:

With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviours of your system will be depended on by somebody.

Use defp.

Loose Coupling

Components in a loosely coupled system can be replaced with alternative implementations that provide the same services.

Polymorphism

The ability to present the same interface for differing underlying forms

Messages

Processes can send messages to other processes without needing to care what the receiver process is or does.

The interface is Process IDs and Kernel.send/2

Behaviors

A module that defines a spec which other modules can implement.

```elixir
defmodule Ockam.Vault do
  @moduledoc false

  @type t :: any()
  @type digest :: <<_::32>>

  @doc """
  Computes the SHA-256 digest of the input.
  """
  @callback sha256(vault :: t, input :: binary()) ::
              {:ok, digest :: digest()} | {:error, reason :: any()}
```

Protocols

A group of function headers that a data type must implement if it is considered to be implementing that protocol.

```elixir
defprotocol Size do
  @doc "Calculates the size (and not the length!) of a data structure"
  def size(data)
end
```

```
defimpl Size, for: BitString do
  def size(string), do: byte_size(string)
end

defimpl Size, for: Map do
  def size(map), do: map_size(map)
end

defimpl Size, for: Tuple do
  def size(tuple), do: tuple_size(tuple)
end
```

Functions

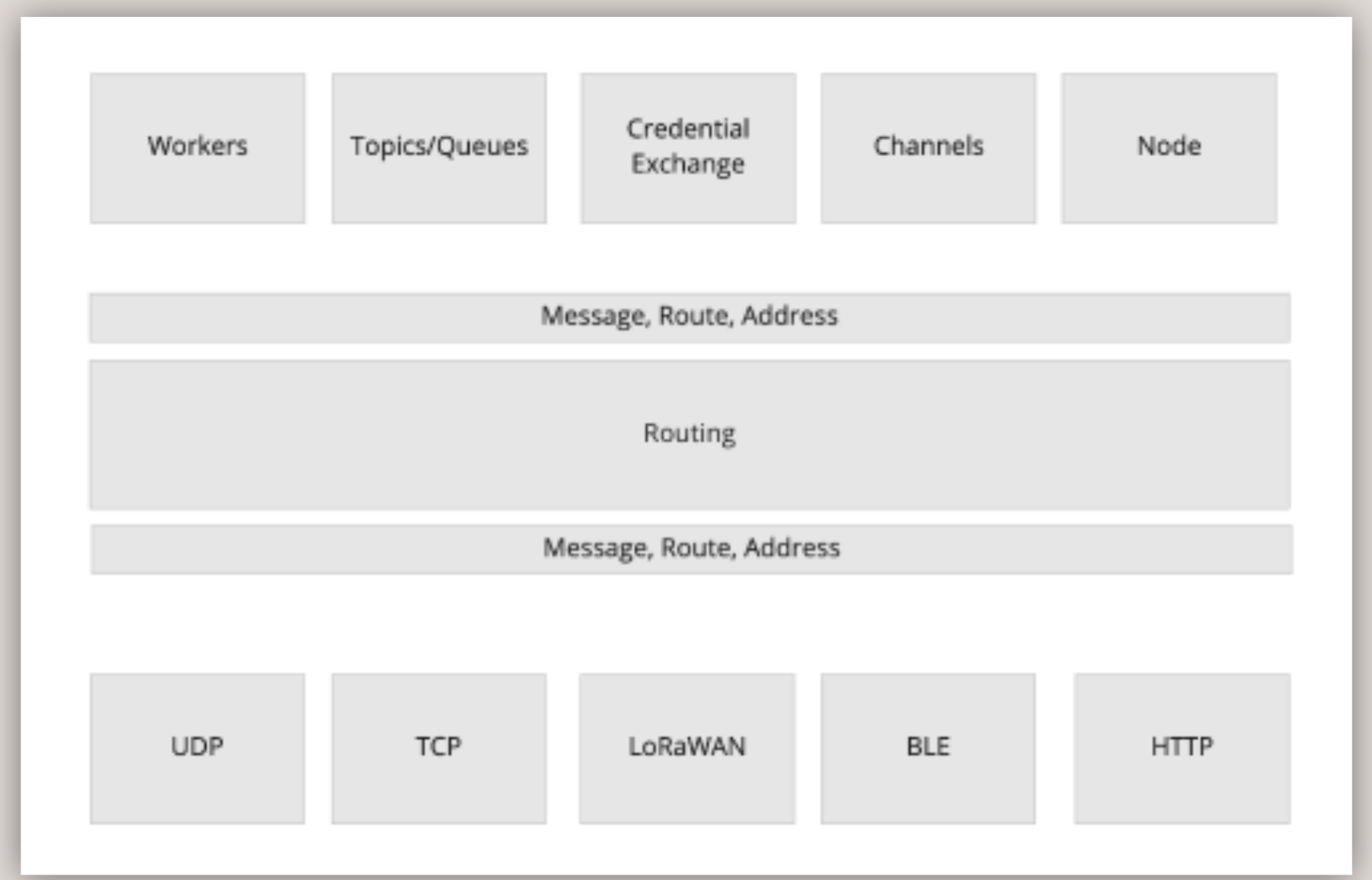
```
iex> Enum.map([1, 2, 3], fn x -> x * 2 end)
[2, 4, 6]

iex> Enum.map([a: 1, b: 2], fn {k, v} -> {k, -v} end)
[a: -1, b: -2]
```

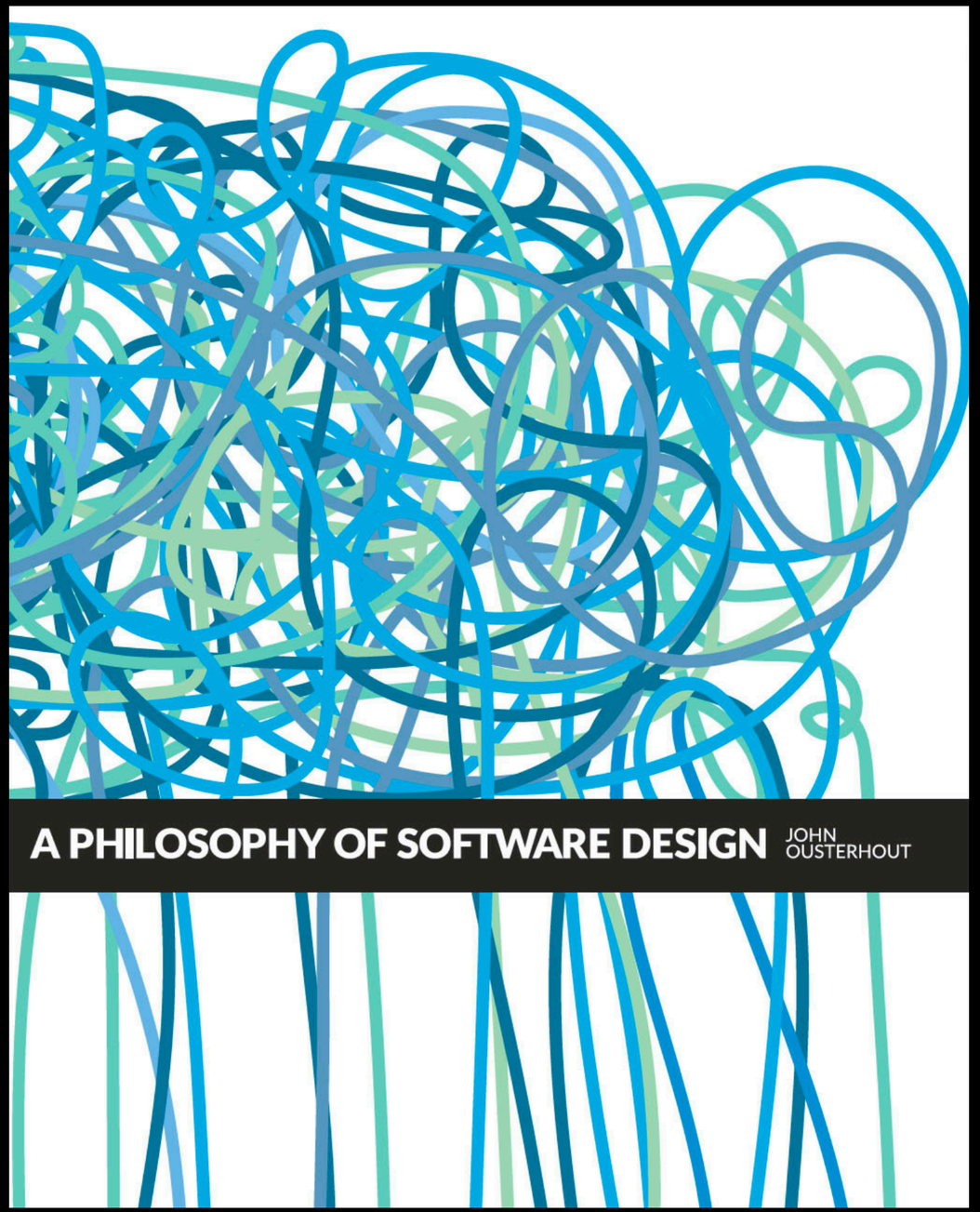
Dependency Inversion

High-level modules should not depend on low-level modules.
Both should depend on abstractions (e.g. interfaces).

Abstractions should not depend on details.
Details (concrete implementations) should depend on abstractions..



github.com/ockam-network/ockam


A PHILOSOPHY OF SOFTWARE DESIGN
JOHN OUSTERHOUT

Mrinal Wadhwa

CTO @ Ockam

twitter.com/mrinal

github.com/ockam-network/ockam