Escape Your Framework

Julian Doherty
@madlep
juliandoherty.com

We’re hiring!

careers.envato.com

I am a lead developer at Envato. We are Melbourne, Australia based company that provides online marketplaces for creatives to buy and sell digital assets.

I think we are hiring, so if moving to Australia appeals, have a chat with me after.

@ElixirMelbourne

https://www.meetup.com/Elixir-Melbourne/

I also founded and run the Elixir Melbourne meetup.

If you’re in town we’d love you to drop in and maybe even do a talk

The brief

- Famous Hollywood movie star wants some feedback on their code
- Working on a startup idea
- Question + Answer site

http://thanksoverflow.com

Needs logo

themoviedb.org

Credits

curl “https://api.themoviedb.org/3/person/31/combined_credits?api_key=YOUR_KEY"


```
{
  "cast": [
    {
      "id": 13,
      "character": "Forrest Gump",
      "original_title": "Forrest Gump",
      "overview": "A man with a low IQ has accomplished great things in his life...",
      "vote_count": 16820,
      "video": false,
      "media_type": "movie",
      "poster_path": "\/yE5d3BUhE8hCnkMUJOo1QDoOGNz.jpg",
      "backdrop_path": "\/tNVTWNa8cxZLw7aWvHzSDLyc7ig.jpg",
      "popularity": 32.293,
      "title": "Forrest Gump",
      "original_language": "en",
      "genre_ids": [
        35,
        18,
        10749
      ],
      "vote_average": 8.4,
      "adult": false,
      "release_date": "1994-07-06",
      "credit_id": "52fe420ec3a36847f800074f"
    },
```

Version 1: Quick And Dirty

OK. This works.

But you need to escape your framework

Version 1: Quick And Dirty

```ruby
class V01::AdminController < ActionController::Base
  def index
  end

  def import_credits
    thanks_id = 31
    api_key = “YOUR_API_KEY_ABC123”
    url = "https://api.themoviedb.org/3/person/#{thanks_id}/combined_credits?api_key=#{api_key}&language=en-US"
    inserted_count = updated_count = error_count = 0
    JSON.parse(HTTP.get(url).to_s)["cast"].each do |cast_entry|
      title = Title.where(tmdb_id: cast_entry["id"]).first
      if title
        title.update( popularity: cast_entry["popularity"], synced_at: DateTime.now)
        updated_count += 1
      else
        title = Title.new(
          tmdb_id: cast_entry["id"],
          title: cast_entry["title"],
          character: cast_entry["character"],
          release_date: cast_entry["release_date"],
          media_type: cast_entry["media_type"],
          popularity: cast_entry["popularity"],
          synced_at: DateTime.now
        )
        if title.valid?
          title.save!
          inserted_count += 1
        else
          error_count += 1
        end
      end
    end
    flash[:notice] = "Imported #{inserted_count} new credits, updated #{updated_count}"
    flash[:error] = "#{error_count} titles had errors preventing saving" if error_count > 0
    redirect_to action: :index
  end
end
```

OK. This works.

But you need to escape your framework

Version 1: Quick And Dirty

- Code coupled to Rails controllers
- Hard coded config
- Hard coded secrets 😱
- Difficult to test
- Difficult to reuse
- Difficult to debug
- Let’s escape the framework...

OK. This works.

But you need to escape your framework

Version 2: Extract From Controller

Version 2: Extract From Controller

```
Rails.application.configure do
  # ...
  config.tmdb_api_key = ENV[“TMDB_API_KEY"]
  config.tmdb_api_url =“https://longurl...“
end
```

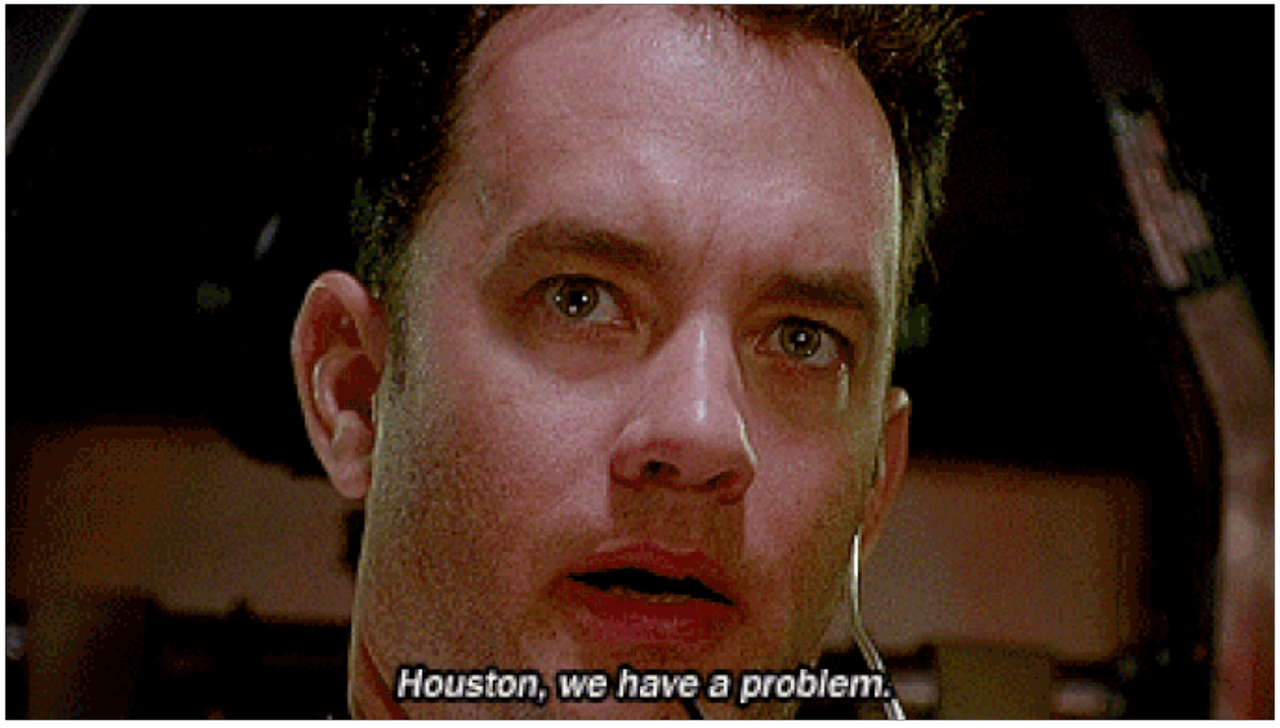
```
class V02::AdminController < ActionController::Base
  THANKS_ID = 31

  def index
  end

  def import_credits
    importer = CreditImporter.new(
      person_id: THANKS_ID,
      url: Rails.configuration.tmdb_api_url
    )
    inserted_count, updated_count, error_count = importer.import()

    flash[:notice] = "Imported #{inserted_count} new credits, updated #{updated_count}"
    flash[:error] = "#{errors_count} titles had errors preventing saving" if error_count > 0
    redirect_to action: :index
  end
end
```

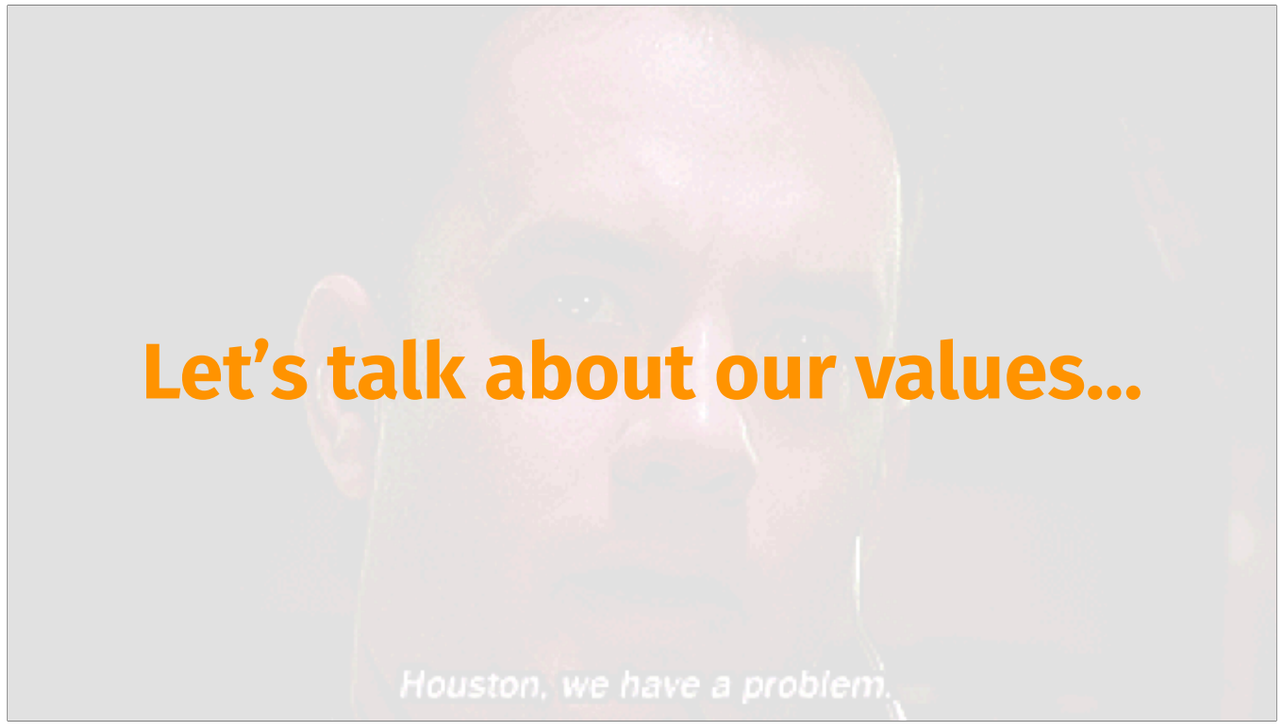
Version 2: Extract From Controller

```ruby
class V02::CreditImporter
  def initialize(person_id:, url:)
    @person_id = person_id
    @url = url
  end

  def import()
    inserted_count = updated_count = error_count = 0
    JSON.parse(HTTP.get(@url).to_s)[“cast”].each do |cast_entry|
      title = Title.where(tmdb_id: cast_entry["id"]).first
      if title
        title.update( popularity: cast_entry["popularity"], synced_at: DateTime.now)
        updated_count += 1
      else
        title = Title.new(
          tmdb_id: cast_entry["id"],
          title: cast_entry["title"],
          character: cast_entry["character"],
          release_date: cast_entry["release_date"],
          media_type: cast_entry["media_type"],
          popularity: cast_entry["popularity"],
          synced_at: DateTime.now
        )
        if title.valid?
          title.save!
          inserted_count += 1
        else
          error_count += 1
        end
      end
    end
    [inserted_count, updated_count, error_count]
  end
end
```

Version 2: Extract From Controller

- Better, but…
- Error cases not handled
- Let’s escape the framework…


Houston, we have a problem.


Let’s talk about our values...
Houston, we have a problem.

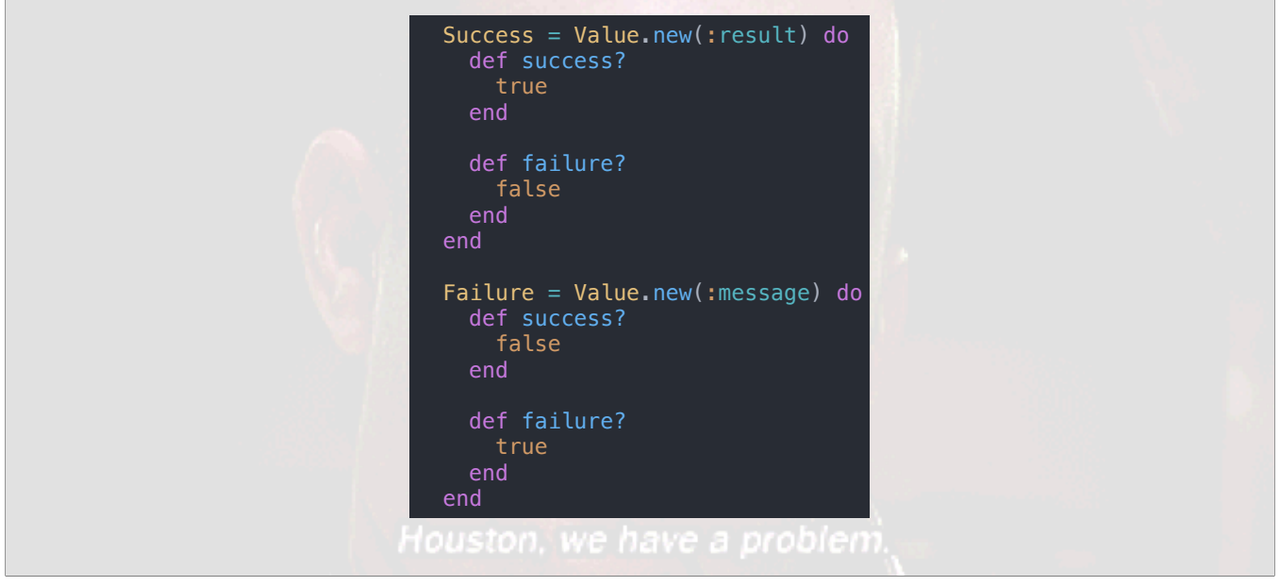
Values

- https://github.com/tcrayford/Values
- https://github.com/madlep/Values/tree/ruby_27_pattern_matching_support

```ruby
Point = Value.new(:x, :y)

some_point = Point.new(123, 456)
#<Point x=123, y=456>

changed_point = some_point.with(y: 789)
#<Point x=123, y=789>

some_point.inspect
#<Point x=123, y=456>
# Hasn't changed!
```

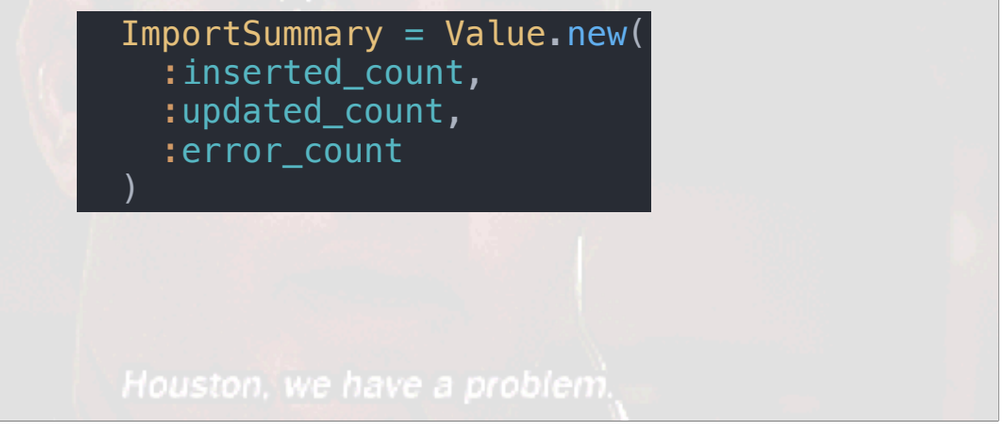
Version 3: Handle errors

```
Success = Value.new(:result) do
  def success?
    true
  end

  def failure?
    false
  end
end

Failure = Value.new(:message) do
  def success?
    false
  end

  def failure?
    true
  end
end
```



Version 3: Handle errors

```
ImportSummary = Value.new(
  :inserted_count,
  :updated_count,
  :error_count
)
```



Version 3: Handle errors

```ruby
def fetch_credits()
  response = HTTP.get(@url)
  if response.status.success?
    Success.new(response.to_s)
  else
    Failure.new("error fetching credits. Status=#{response.status}")
  end
rescue HTTP::Error => e
  Failure.new(e.message)
end



def parse_credits(credits_json)
  begin
    Success.new(JSON.parse(credits_json))
  rescue JSON::ParserError => e
    Failure.new("error parsing credits JSON: '#{e.message}'")
  end
end
```

Version 3: Handle errors

```ruby
def create_or_update_title(cast_entry, import_summary)
  title = Title.where(tmdb_id: cast_entry["id"]).first
  if title
    title.update( popularity: cast_entry["popularity"], synced_at: DateTime.now)
    import_summary.with(updated_count: import_summary.updaated_count + 1)
  else
    title = Title.new(
      tmdb_id: cast_entry["id"],
      title: cast_entry["title"],
      character: cast_entry["character"],
      release_date: cast_entry["release_date"],
      media_type: cast_entry["media_type"],
      popularity: cast_entry["popularity"],
      synced_at: DateTime.now
    )
    if title.valid?
      title.save!
      import_summary.with(inserted_count: import_summary.inserted_count + 1)
    else
      import_summary.with(erro_count: import_summary.error_count + 1)
    end
  end
end
```

Version 3: Handle errors

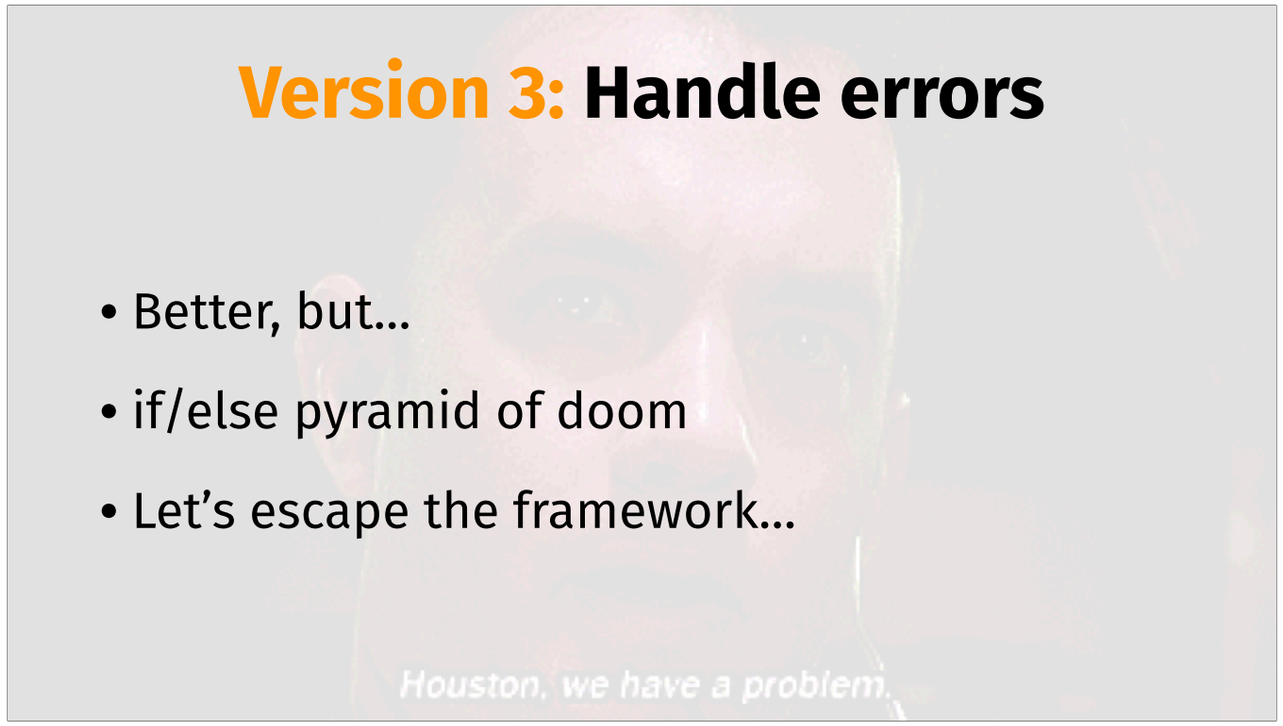
```
def import()
  fetch_credits_result = fetch_credits()
  if fetch_credits_result.failure?
    return fetch_credits_result
  else
    parse_credits_result = parse_credits(fetch_credits_result.result)
    if parse_credits_result.failure?
      return parse_credits_result
    else
      import_summary = ImportSummary.new(0,0,0)
      parse_credits_result.result["cast"].each do |cast_entry|
        import_summary = create_or_update_title(cast_entry, import_summary)
      end
      Success.new(import_summary)
    end
  end
end
```

Houston, we have a problem.

if/else pyramid of doom

Version 3: Handle errors

- Better, but…
- if/else pyramid of doom
- Let’s escape the framework…

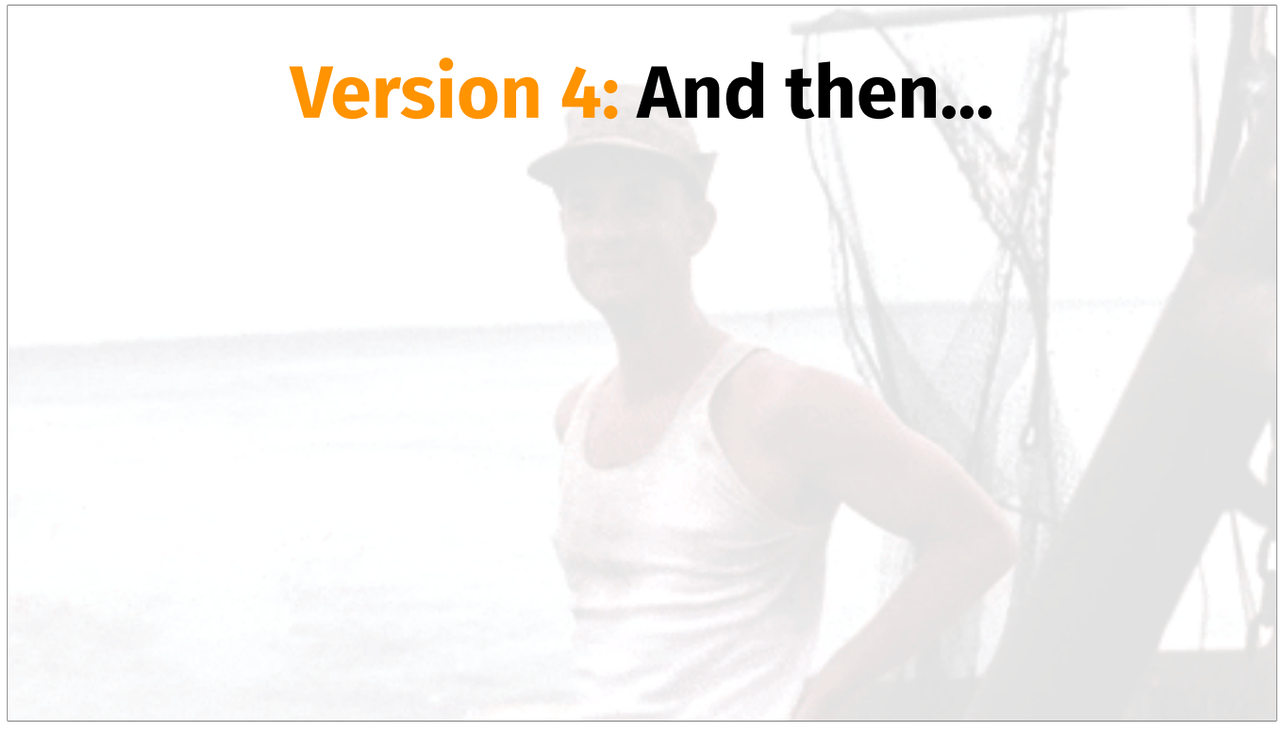


if/else pyramid of doom

Version 4: And then...

```
Success = Value.new(:result) do
  def and_then(&callback)
    callback.(self.result)
  end
end

Failure = Value.new(:message) do
  def and_then(&_callback)
    self
  end
end
```

Version 4: And then...



Version 4: And then...

```
ImportSummary = Value.new(:inserted_count, :updated_count, :error_count) do
  def inserted()
    self.with(inserted_count: self.inserted_count + 1)
  end

  def updated()
    self.with(updated_count: self.updated_count + 1)
  end

  def errored()
    self.with(error_count: self.error_count + 1)
  end
end
```

Version 4: And then...

```
def import()
  fetch_credits()
    .and_then(&method(:parse_credits))
    .and_then(){|credits|
      credits["cast"]
      .reduce(ImportSummary.new(0,0,0), &method(:create_or_update_title))
      .then(){|final_summary|
        Success.new(final_summary)
      }
    }
end
```

Hang on… &method ?

Version 4: And then...

Wait what?

```
.and_then(&method(:parse_credits))
```

Version 4: And then...

```ruby
def square(x)
  x * x
end

square(4)
# 16

[1,2,3,4].map{|x| square(x)}
# [1, 4, 9, 16]
```

Version 4: And then...

```ruby
def square(x)
  x * x
end

squarer = method(:square)
squarer.call(4)
# 16

[1,2,3,4].map{|x| squarer.(x) }
# [1, 4, 9, 16]

[1,2,3,4].map(&squarer)
# [1, 4, 9, 16]

[1,2,3,4].map(&method(:square))
# [1, 4, 9, 16]
```

Version 4: And then...

```
def import()
  fetch_credits()
    .and_then(&method(:parse_credits))
    .and_then(){|credits|
      credits["cast"]
      .reduce(ImportSummary.new(0,0,0), &method(:create_or_update_title))
      .then(){|final_summary|
        Success.new(final_summary)
      }
    }
end
```

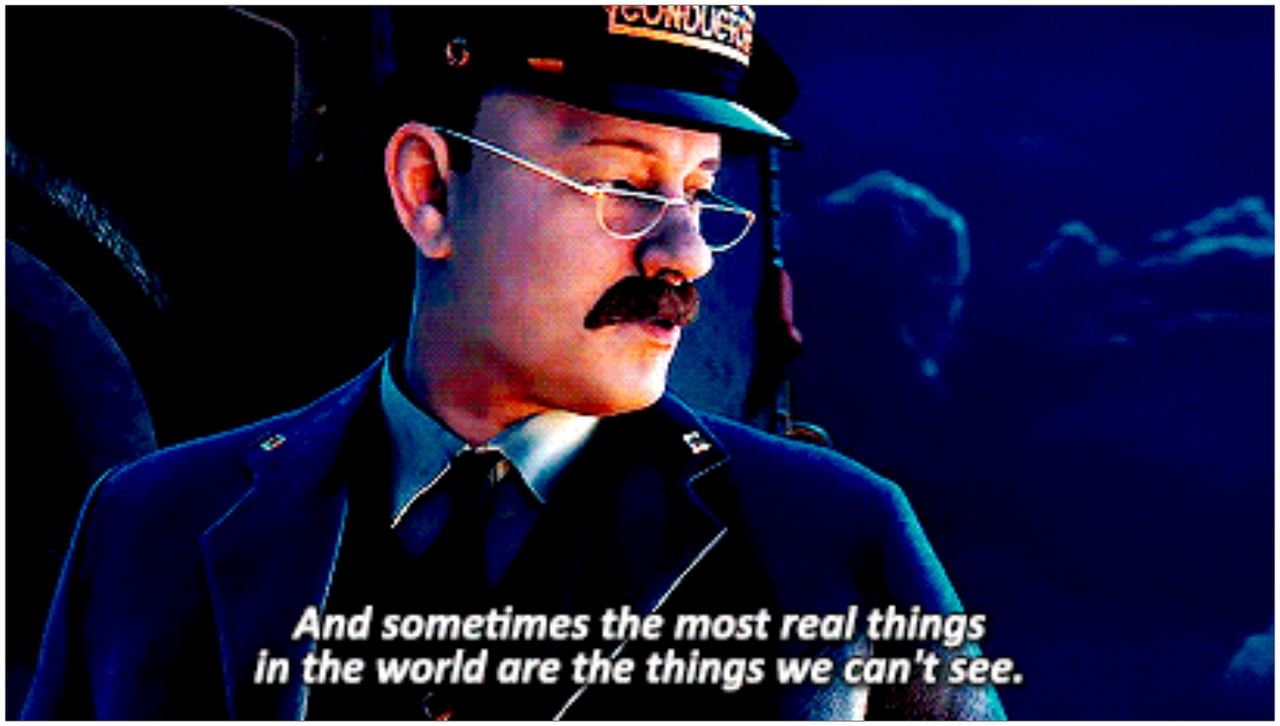
Hang on… &method ?

Version 4: And then...

- Just think about the happy case
- Wrap success/failure in values
- Only proceed on success
- BUT, side effects are all through the logic
- Let's escape the framework...

Houston, we have a problem.

Still riddled with side effects, which makes it harder to test and reuse


And sometimes the most real things
in the world are the things we can't see.

Version 5: Imperative Shell/Functional Core

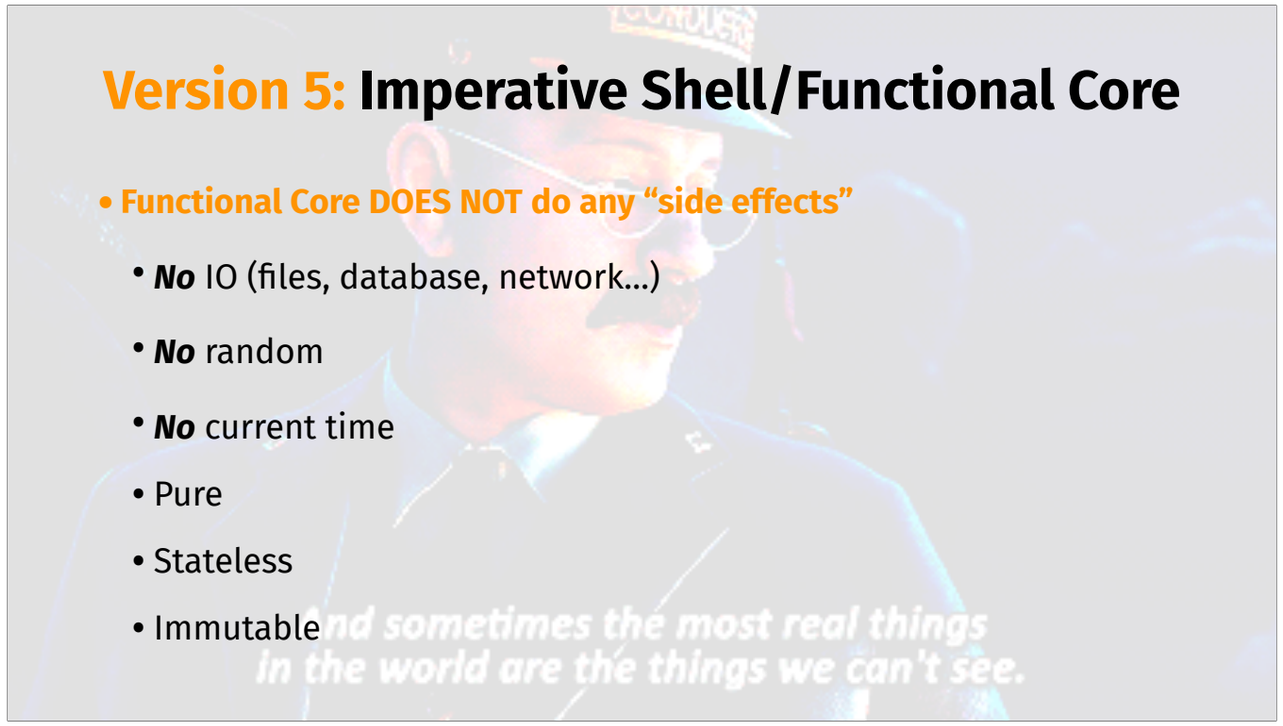
- Imperative state handling in the “shell” layer
- Business logic in the “functional core”
- Shell is simple, provides services, delegates to core
- Core does the work.
- Core DOES NOT do any “side effects”

Version 5: Imperative Shell/Functional Core

- Imperative state handling in the “shell” layer
- Business logic in the “functional core”
- Shell is simple, provides “effectful” services
- Shell delegates to core to do work
- Application uses the shell API
- Core does the work.
- Core DOES NOT do any “side effects”

Version 5: Imperative Shell/Functional Core

- **Functional Core DOES NOT do any “side effects”**
  - ***No*** IO (files, database, network...)
  - ***No*** random
  - ***No*** current time
  - Pure
  - Stateless
  - Immutable

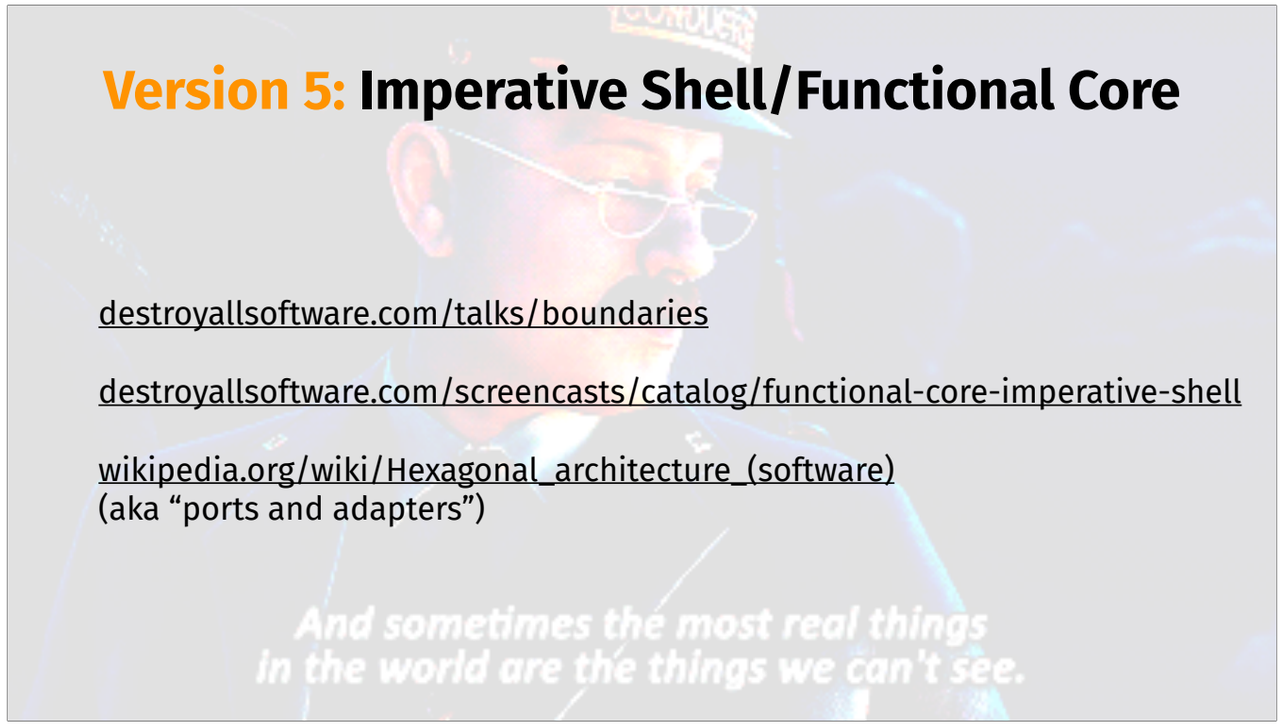


Version 5: Imperative Shell/Functional Core

destroyallsoftware.com/talks/boundaries

destroyallsoftware.com/screencasts/catalog/functional-core-imperative-shell

wikipedia.org/wiki/Hexagonal_architecture_(software)
(aka “ports and adapters”)



Version 5: Imperative Shell/Functional Core

```ruby
class V05::CreditImporterShell
  def import()
    CreditImporterCore.new.(
      method(:fetch_credits),
      method(:find_title),
      ImportSummary.new(0,0,0)
    )
  end

  private
  def fetch_credits()
    # ...
  end

  def find_title(tmdb_id)
    # ...
  end
end
```

Inject methods into functional core so it doesn’t have to handle that logic.
Shell provides effectful services

Version 5: Imperative Shell/Functional Core

```ruby
class V05::CreditImporterCore
  def call(fetch_credits, find_title, import_summary)
    fetch_credits.()
      .and_then(&method(:parse_credits))
      .and_then(&method(:import_credits).curry.(import_summary, find_title))
  end

  def parse_credits(credits_json)
    # ...
  end

  def import_credits(import_summary, find_title, credits)
    credits["cast"]
    .reduce(import_summary, &method(:create_or_update_title).curry.(find_title))
    .then{|final_summary|
      Success.new(final_summary)
    }
  end

  def create_or_update_title(find_title, import_summary, cast_entry)
    title = find_title.(cast_entry["id"])
    if title
      # ...
    else
      # ...
    end
  end
end
```

Version 5: Imperative Shell/Functional Core

```ruby
def call(fetch_credits, find_title, import_summary)
  fetch_credits.()
    .and_then(&method(:parse_credits))
    .and_then(&method(:import_credits).curry.(import_summary, find_title))
end
```

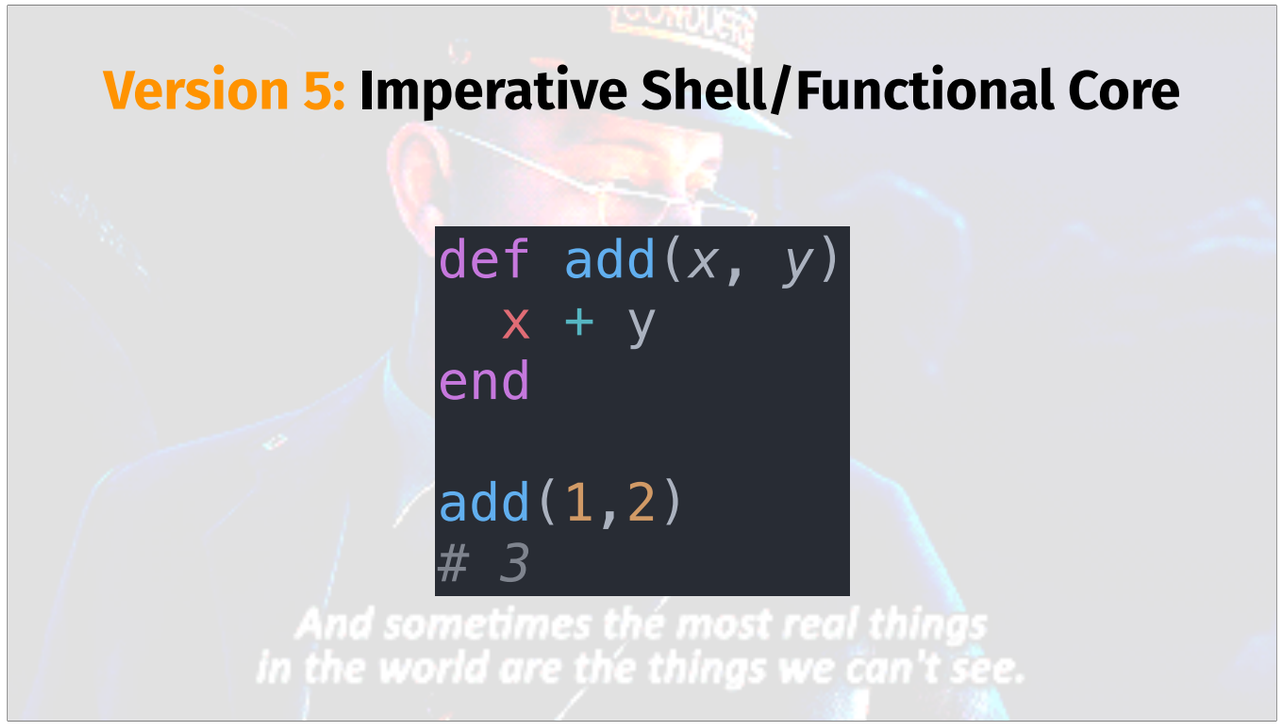
Version 5: Imperative Shell/Functional Core

Wait what?

```
&method(:import_credits).curry.(import_summary, find_title))
```

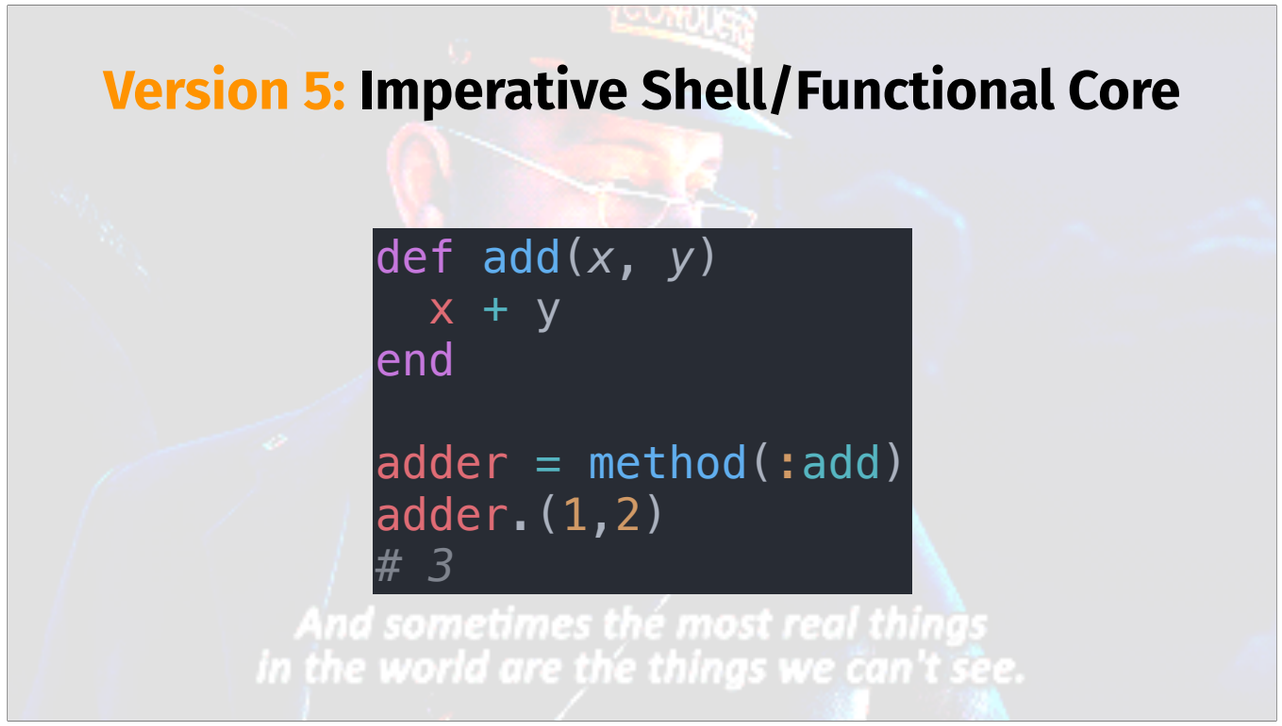
Version 5: Imperative Shell/Functional Core

```ruby
def add(x, y)
  x + y
end

add(1,2)
# 3
```



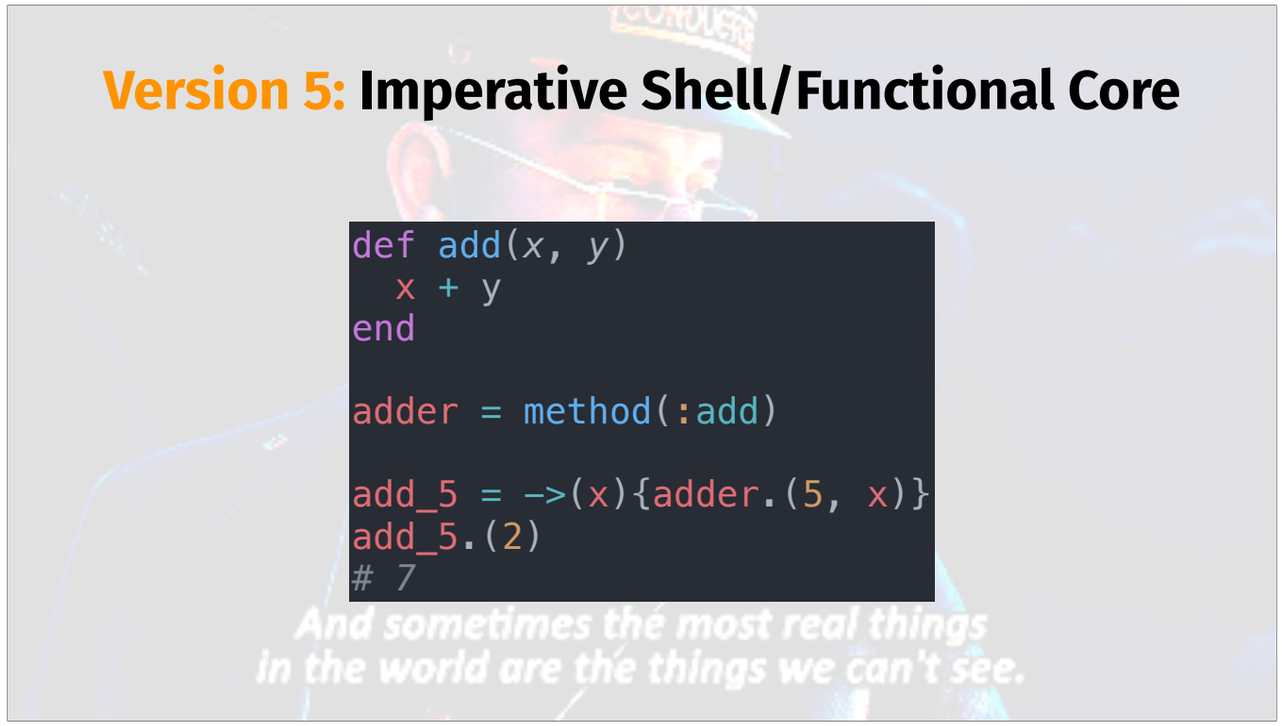
Version 5: Imperative Shell/Functional Core

```ruby
def add(x, y)
  x + y
end

adder = method(:add)
adder.(1,2)
# 3
```



Version 5: Imperative Shell/Functional Core

```ruby
def add(x, y)
  x + y
end

adder = method(:add)

add_5 = ->(x){adder.(5, x)}
add_5.(2)
# 7
```



Version 5: Imperative Shell/Functional Core

```ruby
def add(x, y)
  x + y
end

add_5 = adder.curry.(5)
add_5.(2)
# 7

add_5 = method(:add).curry.(5)
add_5.(2)
# 7
```

Version 5: Imperative Shell/Functional Core

```
def call(fetch_credits, find_title, import_summary)
  fetch_credits.()
    .and_then(&method(:parse_credits))
    .and_then(&method(:import_credits).curry.(import_summary, find_title))
end
```

Version 5: Imperative Shell/Functional Core


```ruby
def import_credits(import_summary, find_title, credits)
  credits["cast"]
  .reduce(import_summary, &method(:create_or_update_title).curry.(find_title))
  .then{|final_summary|
    Success.new(final_summary)
  }
end
```

Same thing in main import_credits function. This is the core of what’s going on here. We inject find_title to create_or_update_title. Reduce iterates over all the cast_entries

Version 5: Imperative Shell/Functional Core

```ruby
def create_or_update_title(find_title, import_summary, cast_entry)
  title = find_title.(cast_entry["id"])
  if title
    title.update( popularity: cast_entry["popularity"], synced_at: DateTime.now)
    import_summary.updated()
  else
    title = Title.new(
      tmdb_id: cast_entry["id"],
      title: cast_entry["title"],
      character: cast_entry["character"],
      release_date: cast_entry["release_date"],
      media_type: cast_entry["media_type"],
      popularity: cast_entry["popularity"],
      synced_at: DateTime.now
    )
    if title.valid?
      title.save!
      import_summary.inserted()
    else
      import_summary.errored()
    end
  end
```

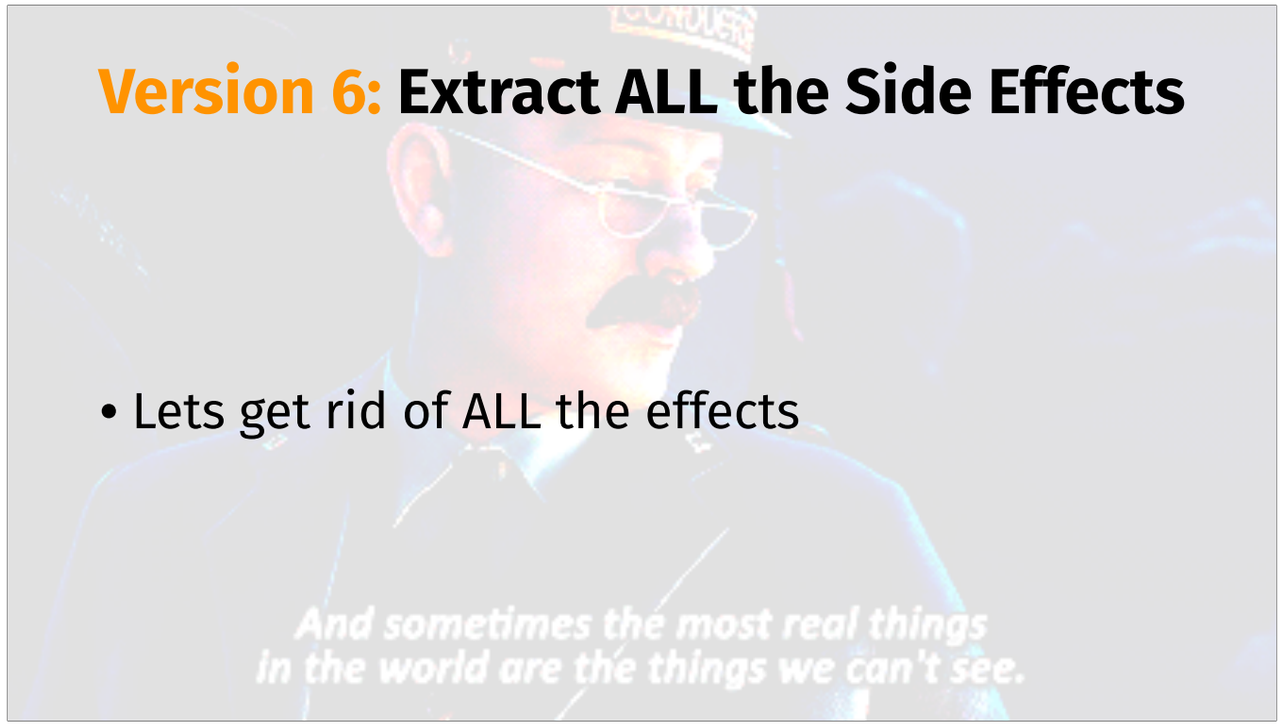
Version 5: Imperative Shell/Functional Core

- Remove effectful code from functional core
- Imperative shell provide services as functions
- Dependency inject services into core
- We've only done *read* side effects so far
- Let's escape the framework...

Houston, we have a problem.

Still riddled with side effects, which makes it harder to test and reuse

Version 6: Extract ALL the Side Effects



- Lets get rid of ALL the effects

Version 6: Extract ALL the Side Effects

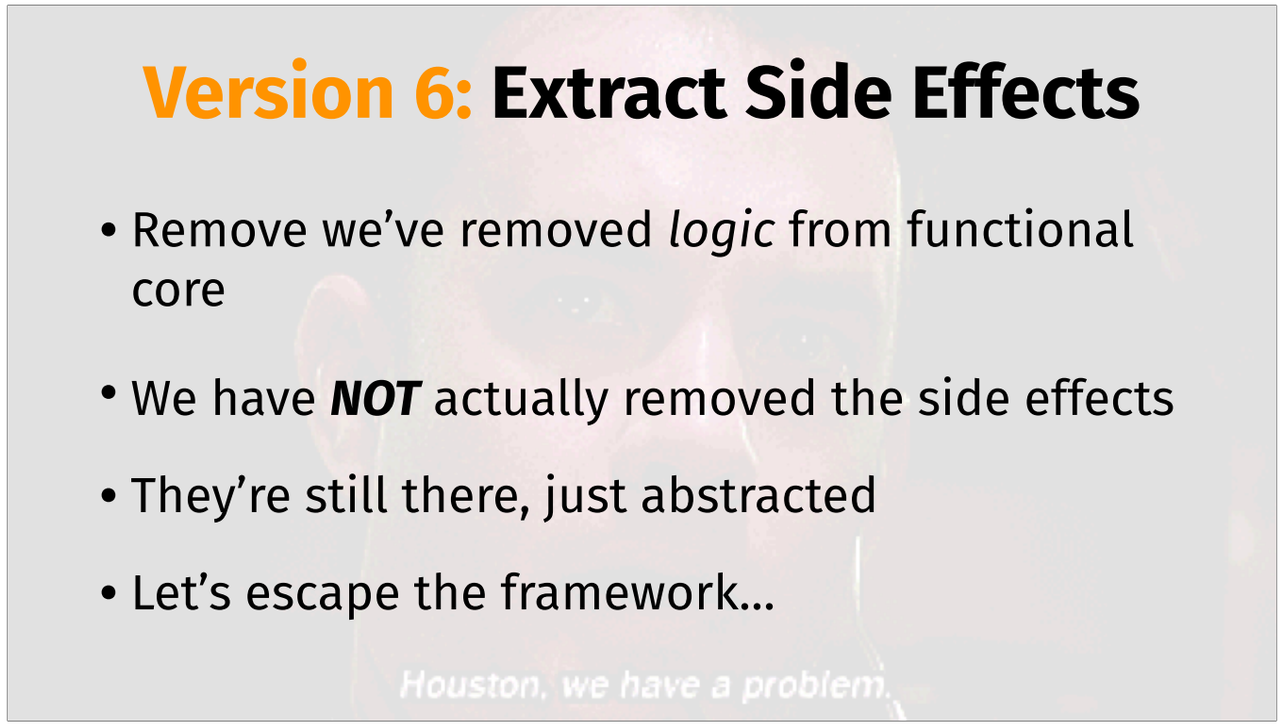
```ruby
class V06::CreditImporterShell
  def import()
    CreditImporterCore.new.(
      method(:fetch_credits),
      method(:find_title),
      method(:insert_title),
      method(:update_title),
      ImportSummary.new(0,0,0)
    )
  end

  private
  def fetch_credits()
    # ...
  end

  def find_title(tmdb_id)
    # ...
  end

  def insert_title(import_summary, **insert_attrs)
    # ...
  end

  def update_title(import_summary, title, **update_attrs)
    # ...
  end
end
```

Version 6: Extract ALL the Side Effects

```
def create_or_update_title(find_title, insert_title, update_title, import_summary, cast_entry)
  title = find_title.(cast_entry["id"])
  if title
    update_title.(
      # ...
    )
  else
    insert_title.(
      # ...
    )
  end
end
```

Version 6: Extract Side Effects

- Remove we've removed *logic* from functional core
- We have ***NOT*** actually removed the side effects
- They're still there, just abstracted
- Let's escape the framework...



Still riddled with side effects, which makes it harder to test and reuse

Version 7: Return write side effect values

```
InsertTitle = Value.new(:insert_attrs)
UpdateTitle = Value.new(:title, :update_attrs)
```

Version 7: Return write side effect values

```ruby
class V07::CreditImporterCore
  def import_credits(find_title, credits)
    credits["cast"]
    .reduce([]) { |actions, cast_entry|
      actions + [create_or_update_title(find_title, cast_entry)]
    }.then{|final_summary|
      Success.new(final_summary)
    }
  end

  def create_or_update_title(find_title, cast_entry)
    title = find_title.(cast_entry["id"])
    if title
      UpdateTitle.new(
        # ...
      )
    else
      InsertTitle.new(
        # ...
      )
    end
  end
end
```

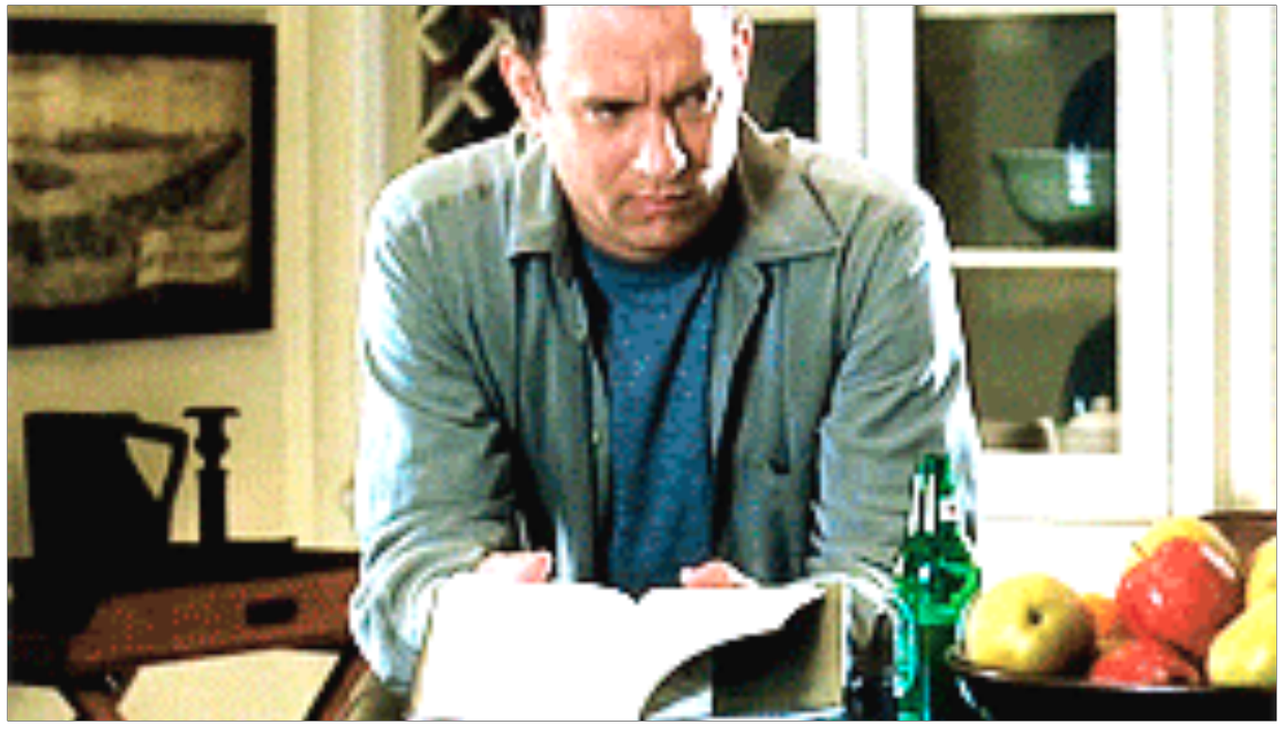
Version 7: Return write side effect values

```ruby
class V07::CreditImporterShell
  def import()
    CreditImporterCore.new.(
      method(:fetch_credits),
      method(:find_title)
    ).and_then(&method(:apply))
  end

  private
  def apply(actions)
    actions.reduce(ImportSummary.new(0,0,0)){|summary, action|
      case action
      in InsertTitle[insert_attrs]
        insert_title(summary, insert_attrs)
      in UpdateTitle[title, update_attrs]
        update_title(summary, title, update_attrs)
      end
    }.then{|final_summary|
      Success.new(final_summary)
    }
  end
end
```

Version 7: Return write side effect values

- Our functional core no longer changes the world
- Writes are values returned from a function
- Shell interprets values as effects, and executes them
- Still read side effects to deal with
- Let’s escape the framework...

Still riddled with side effects, which makes it harder to test and reuse



Version 8: Return write side effect values

- Returning functional side effects that *read* is trickier
- How do you “return” something that you had to execute already?
- Let’s put them in a value, with a **`next`** field containing a function to call when we *do* get the value
- Then return that value
- Let’s escape the framework…

Still riddled with side effects, which makes it harder to test and reuse

Version 8: Return read side effect values too

```
FetchCredits = Value.new(:next)

FindTitle = Value.new(:tmdb_id, :next)

InsertTitle = Value.new(:insert_attrs, :next)

UpdateTitle = Value.new(:title, :update_attrs, :next)

Empty = Value.new(:next)

Return = Value.new(:result)

def pure
  -> (result) { Return.new(result) }
end
```

Version 8: Return read side effect values too

```
module AndThen
  def and_then(&block)
    self.with(next: ->(value) do
      self.next.(value)
        .and_then(&block)
    end)
  end
end
```

Version 8: Return read side effect values too

```
FetchCredits = Value.new(:next) do
  include AndThen
end

FindTitle = Value.new(:tmdb_id, :next) do
  include AndThen
end

InsertTitle = Value.new(:insert_attrs, :next) do
  include AndThen
end

UpdateTitle = Value.new(:title, :update_attrs, :next) do
  include AndThen
end

Empty = Value.new(:next) do
  include AndThen
end

Return = Value.new(:result) do
  def and_then(&block)
    block.call(self.result)
  end
end

def pure
  -> (result) { Return.new(result) }
end
```

Version 8: Return read side effect values too

```
class V08::CreditImporterCore
  def call()
    FetchCredits.new(->(credits_json){
      parse_credits(credits_json)
      .and_then(&method(:import_credits))
    })
  end

  def import_credits(credits)
    credits["cast"]
    .reduce(Empty.new(pure)) { |previous_action, cast_entry|
      previous_action.and_then {|_value|
        create_or_update_title(cast_entry)
      }
    }
  end

  def create_or_update_title(cast_entry)
    FindTitle.new(cast_entry["id"], -> (title){
      if title
        UpdateTitle.new(title, {...}, pure)
      else
        InsertTitle.new({...}, pure)
      end
    })
  end
end
```

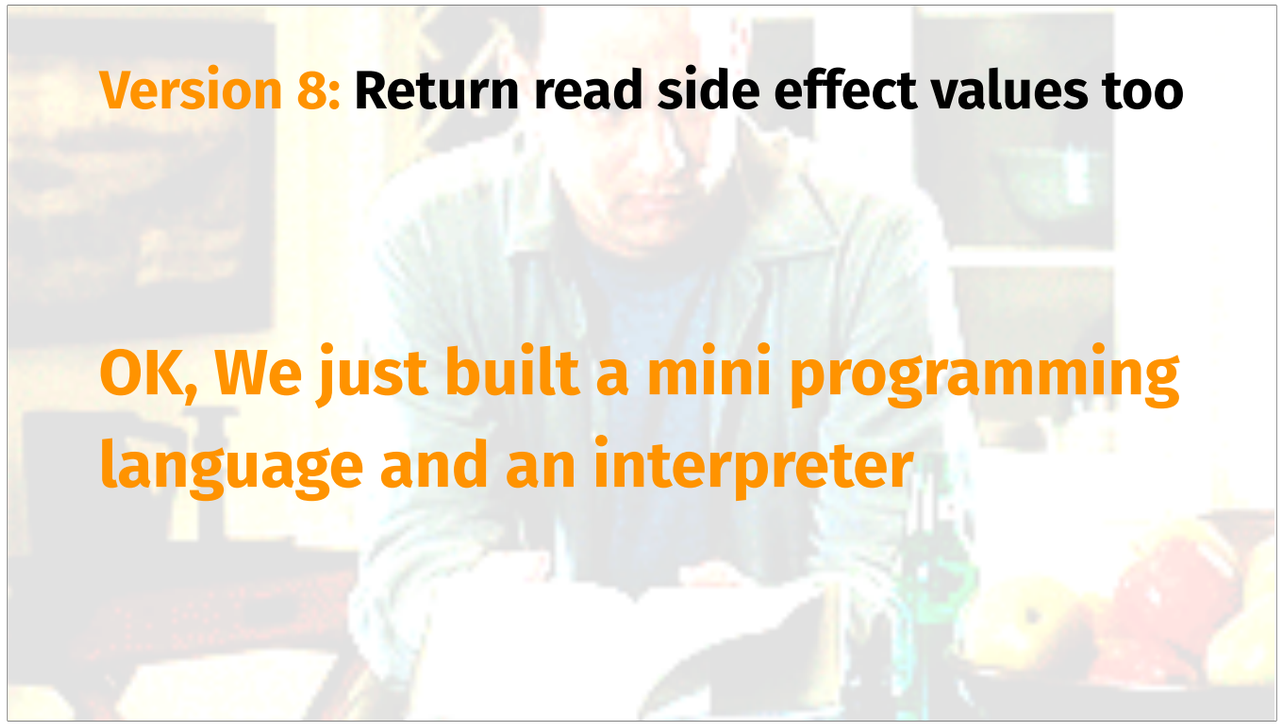
Version 8: Return read side effect values too

```
class V08::CreditImporterShell
  def import()
    CreditImporterCore.().then{|action|
      _result, summary = apply(action, ImportSummary.new(0,0,0))
      Success.new(summary)
    }
  end

  private
  def apply(action, summary)
    case action
    in FetchCredits[do_next]
      fetch_credits.and_then{|credits_json|
        apply(do_next.(credits_json), summary)
      }
    in FindTitle[tmdb_id, do_next]
      title = find_title(tmdb_id)
      apply(do_next.(title), summary)
    in InsertTitle[insert_attrs, do_next]
      new_summary = insert_title(summary, insert_attrs)
      apply(do_next.(:ok), new_summary)
    in UpdateTitle[title, update_attrs, do_next]
      new_summary = update_title(summary, title, update_attrs)
      apply(do_next.(:ok), new_summary)
    in Empty[do_next]
      apply(do_next.(:empty), summary)
    in Return[result]
      [result, summary]
    end
  end
end
```

What we've done here is implemented an interpreter.

Our core generates a program - basically an AST.

Our shell executes it, and takes care of talking to all the stateful things in the world



What we've done here is implemented an interpreter.

Our core generates a program - basically an AST.

Our shell executes it, and takes care of talking to all the stateful things in the world

Version 8: Return read side effect values too

- Reads can be encapsulated with an optional value, and a next operation to return
- Read values can be chained
- In fact you *have* to chain them, as they need to be returned (side effects don't just "happen" anymore)
- API is still clunky
- Let's escape the framework...

Still riddled with side effects, which makes it harder to test and reuse

Version 9: Nicer API

- Let’s get a nicer API that is more rubyish
- Let’s escape the framework...

Version 9: Nicer API

```ruby
def fetch_credits()
  FetchCredits.new(pure)
end

def find_title(tmdb_id)
  FindTitle.new(tmdb_id, pure)
end

def insert_title(**insert_attrs)
  InsertTitle.new(insert_attrs, pure)
end

def update_title(title, **update_attrs)
  UpdateTitle.new(title, update_attrs, pure)
end

def empty()
  Empty.new(pure)
end

def do_return(result)
  Return.new(result)
end
```

Version 9: Nicer API

```
class V09::CreditsImporterCore
  def call()
    fetch_credits()
    .and_then(&method(:parse_credits))
    .and_then(&method(:import_credits))
  end

  def parse_credits(credits_json)
    # ...
  end

  def import_credits(credits)
    # ...
  end

  def create_or_update_title(cast_entry)
    find_title(cast_entry["id"]).and_then {|title|
      if title
        update_title(title, {...})
      else
        insert_title({...}
      end
    }
  end
end
```

Version 9: Nicer API

```
class V09::CreditImporterShell
  def import()
    ImportCredits.new.().then{|action|
      _result, summary = apply(action, ImportSummary.new(0,0,0))
      success(summary)
    }
  end

  private
  def apply(action, summary)
    case action
    in FetchCredits[do_next]
      fetch_credits.and_then{|credits_json|
        apply(do_next.(credits_json), summary)
      }
    in FindTitle[tmdb_id, do_next]
      title = find_title(tmdb_id)
      apply(do_next.(title), summary)
    in InsertTitle[insert_attrs, do_next]
      new_summary = insert_title(summary, insert_attrs)
      apply(do_next.(:ok), new_summary)
    in UpdateTitle[title, update_attrs, do_next]
      new_summary = update_title(summary, title, update_attrs)
      apply(do_next.(:ok), new_summary)
    in Empty[do_next]
      apply(do_next.(:empty), summary)
    in Return[result]
      [result, summary]
    end
  end
end
```

Version 1: Quick And Dirty

```ruby
class V01::AdminController < ActionController::Base
  def index
  end

  def import_credits
    thanks_id = 31
    api_key = “YOUR_API_KEY_ABC123”
    url = "https://api.themoviedb.org/3/person/#{thanks_id}/combined_credits?api_key=#{api_key}&language=en-US"
    inserted_count = updated_count = error_count = 0
    JSON.parse(HTTP.get(url).to_s)["cast"].each do |cast_entry|
      title = Title.where(tmdb_id: cast_entry["id"]).first
      if title
        title.update( popularity: cast_entry["popularity"], synced_at: DateTime.now)
        updated_count += 1
      else
        title = Title.new(
          tmdb_id: cast_entry["id"],
          title: cast_entry["title"],
          character: cast_entry["character"],
          release_date: cast_entry["release_date"],
          media_type: cast_entry["media_type"],
          popularity: cast_entry["popularity"],
          synced_at: DateTime.now
        )
        if title.valid?
          title.save!
          inserted_count += 1
        else
          error_count += 1
        end
      end
    end
    flash[:notice] = "Imported #{inserted_count} new credits, updated #{updated_count}"
    flash[:error] = "#{error_count} titles had errors preventing saving" if error_count > 0
    redirect_to action: :index
  end
end
```

OK. This works.

But you need to escape your framework

Version 9: Nicer API

```
class V09::CreditsImporterCore
  def call()
    fetch_credits()
    .and_then(&method(:parse_credits))
    .and_then(&method(:import_credits))
  end

  def parse_credits(credits_json)
    # ...
  end

  def import_credits(credits)
    # ...
  end

  def create_or_update_title(cast_entry)
    find_title(cast_entry["id"]).and_then {|title|
      if title
        update_title(title, {...})
      else
        insert_title({...}
      end
    }
  end
end
```

BONUS SUPRISE! : Monads!

```
Success = Value.new(:result) do
  def and_then(&callback)
    callback.(self.result)
  end
end

Failure = Value.new(:message) do
  def and_then(&_callback)
    self
  end
end
```

BONUS SUPRISE! : Monads!

- Chaining commands based on success/ failure?

  `Result` *(aka* `Either`*) Monad*

BONUS SUPRISE! : Monads!

- Building a list of effectful values: ***Free Monad***
- ***Matt Parsons - I Command You to be Free! - λC 2017***
  youtube.com/watch?v=Ej5FQtEgTBw

Thank You!

Julian Doherty
@madlep
juliandoherty.com

And to find out when you can play it! follow me on twitter, or my site.

Thank you!

Components can live where ever

```
@components = {
  position: {},
  velocity: {},
  acceleration: {},
  sad_feeling: {}
}

def put_component(type, entity_id, component)
  @components[type][entity_id] = component
end

asteroid_id = 123
put_component(:position, asteroid_id, Vector2.new(x: 1, y: 2))
put_component(:velocity, asteroid_id, Vector2.new(x: 11, y: 22))
put_component(:acceleration, asteroid_id, Vector2.new(x: 101, y: 202))
```

So let’s look at how that would work in practice.

Apologies here, as I’m showing this in Ruby to set up a straw man to beat up. Same thing can be done in Elixir, but it’s a little clearer explaining it this way.

So we have a map of all our component stores. One for each type. To create a game object, we just put a bunch of components in different stores for that entity id.

Systems only care about what they care about

```
def do_movement_system()
  (@components[:position].keys() &
    @components[:velocity].keys() &
    @components[:acceleration].keys()
  ).each do |entity_id|
    acc = @components[:acceleration][entity_id]
    vel = @components[:velocity][entity_id]
    pos = @components[:position][entity_id]

    new_vel = vel + acc
    @components[:velocity][entity_id] = new_vel

    new_pos = pos + new_vel
    @components[:position][entity_id] = new_pos
  end
end
```

Here we just find entity ids that have all the components we care about - position, velocity, acceleration. And then do simple newtonian physics to move them.

Systems only care about what they care about

```
put_component(:sad_feeling, asteroid_id, "feeling sad")

def do_sadness_system()
  (@components[:sad_feeling].keys() &
    @components[:velocity].keys() &
    @components[:acceleration].keys()
  ).each do |entity_id|
    if @components[:sad_feeling][entity_id] == "feeling sad"
      @components[:acceleration].delete(entity_id)
      @components[:velocity].delete(entity_id)
      @components[:sad_feeling][entity_id] = "feeling a bit better”
    end
  end
end
```

Then we can have a separate system handling our sad asteroid. If it’s sad, it just stops.

So systems can read and write components.

Systems can be parallelized

```
Thread.new do
  loop do
    do_movement_system()
    sleep(0.1)
  end
end

Thread.new do
  loop do
    do_sadness_system()
    sleep(0.1)
  end
end
```

And having data split up, means they can be parallelized.

I can’t think of ANYTHING that could POSSIBLY go wrong with concurrent, shared mutable state...

```ruby
def do_movement_system()
  # ...
  @components[:velocity][entity_id] = new_vel
  # ...
end

def do_sadness_system()
  # ...
  @components[:velocity].delete(entity_id)
  # 😱
end
```

They can ge be parallelized… 😯

Now I can’t think of ANYTHING that could POSSIBLY go wrong with concurrent shared mutable state.

😬

Yeah, good luck debugging that when it comes up

Don’t feel too smug... yet

You can screw things up this much in Elixir too with things like ets or databases

You can still mess this up in Elixir too.

First cut of this I did exactly that by backing it with an ETS table. Which seemed like a good idea until I thought about it for a bit that there is no way to do concurrently transactions on an ets table and data gets splatted real quick