Railway Oriented Programming

A functional approach to error handling

Scott Wlaschin
@ScottWlaschin

Examples will be in F#...

fsharpforfunandprofit.com

...but OCaml and Haskell are very similar.

FPbridge.co.uk

Overview

Topics covered:

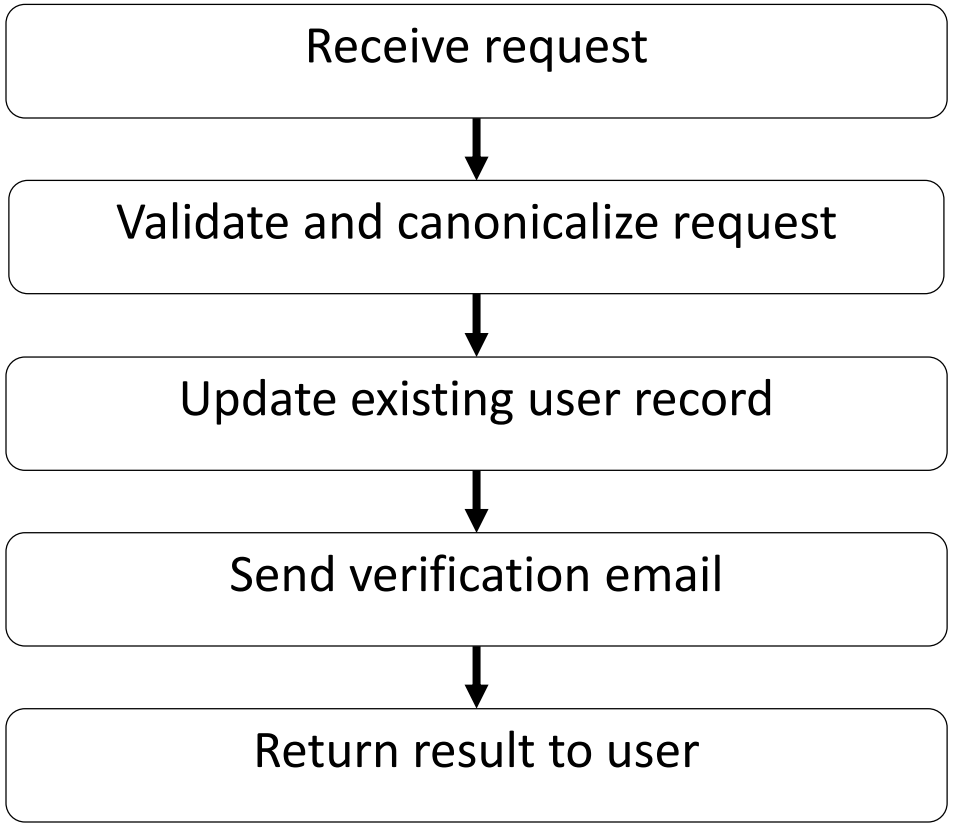
- Happy path programming
- Straying from the happy path
- Introducing "Railway Oriented Programming"
- Using the model in practice
- Extending and improving the design

Happy path programming

Implementing a simple use case

A simple use case

"*As a user I want to update my name and email address*"



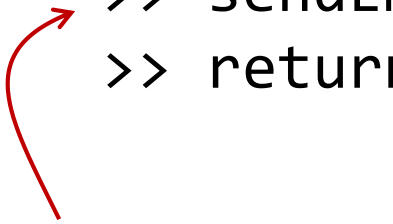

```
type Request = {
  userId: int;
  name: string;
  email: string }
```

Imperative code

```
string ExecuteUseCase()
{
  var request = receiveRequest();
  validateRequest(request);
  canonicalizeEmail(request);
  db.updateDbFromRequest(request);
  smtpServer.sendEmail(request.Email);
  return "Success";
}
```

Functional flow

```
let executeUseCase =
  receiveRequest
  >> validateRequest
  >> canonicalizeEmail
  >> updateDbFromRequest
  >> sendEmail
  >> returnMessage
```



F# left-to-right composition operator

Functional flow (Haskell version)

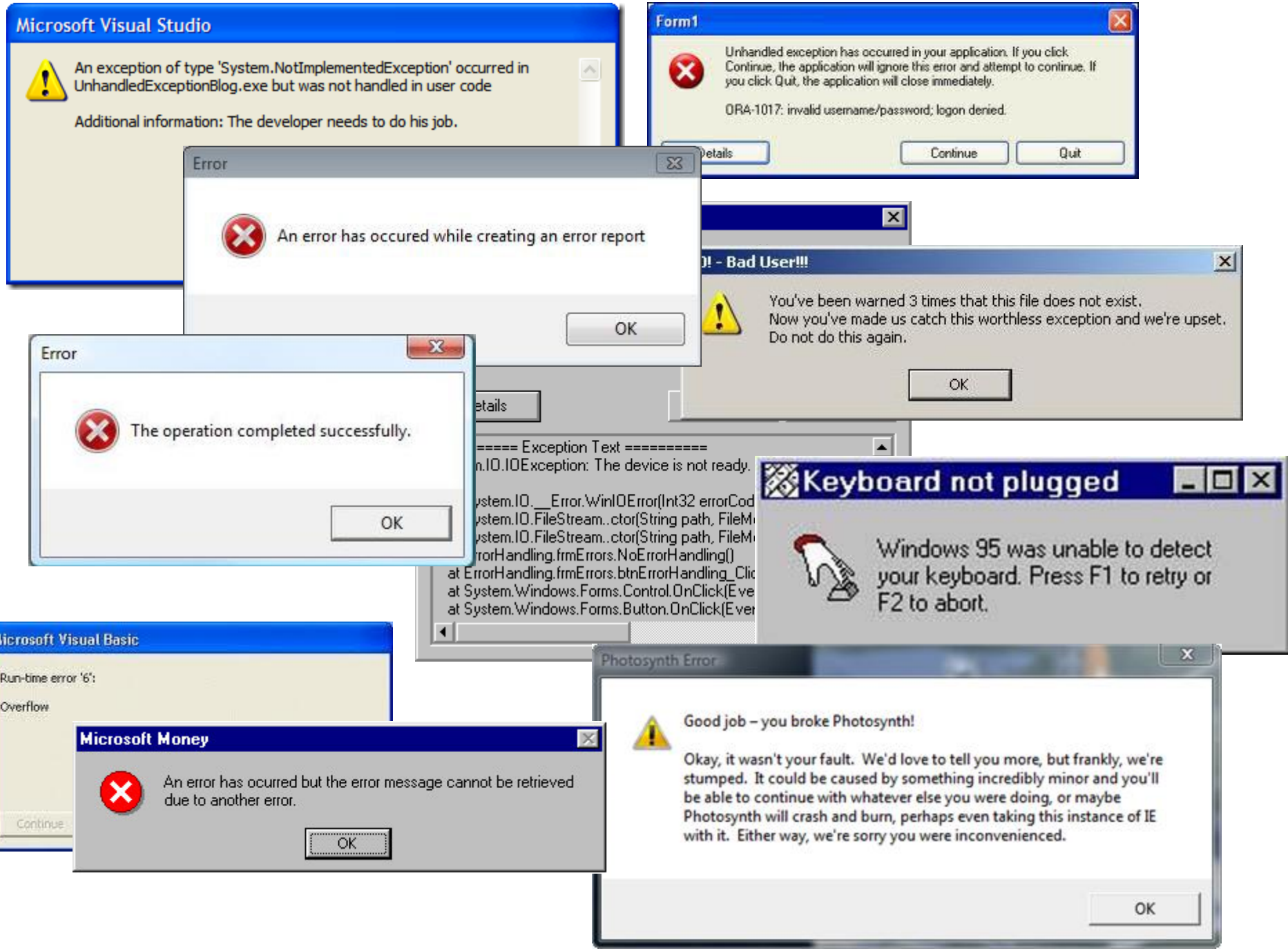
```
let executeUseCase =
  receiveRequest
  >>validateRequest
  >>canonicalizeEmail
  >>updateDbFromRequest
  >>sendEmail
  >>returnMessage
```

Haskell version!

Straying from the happy path...

What do you do when
something goes wrong?

Straying from the happy path

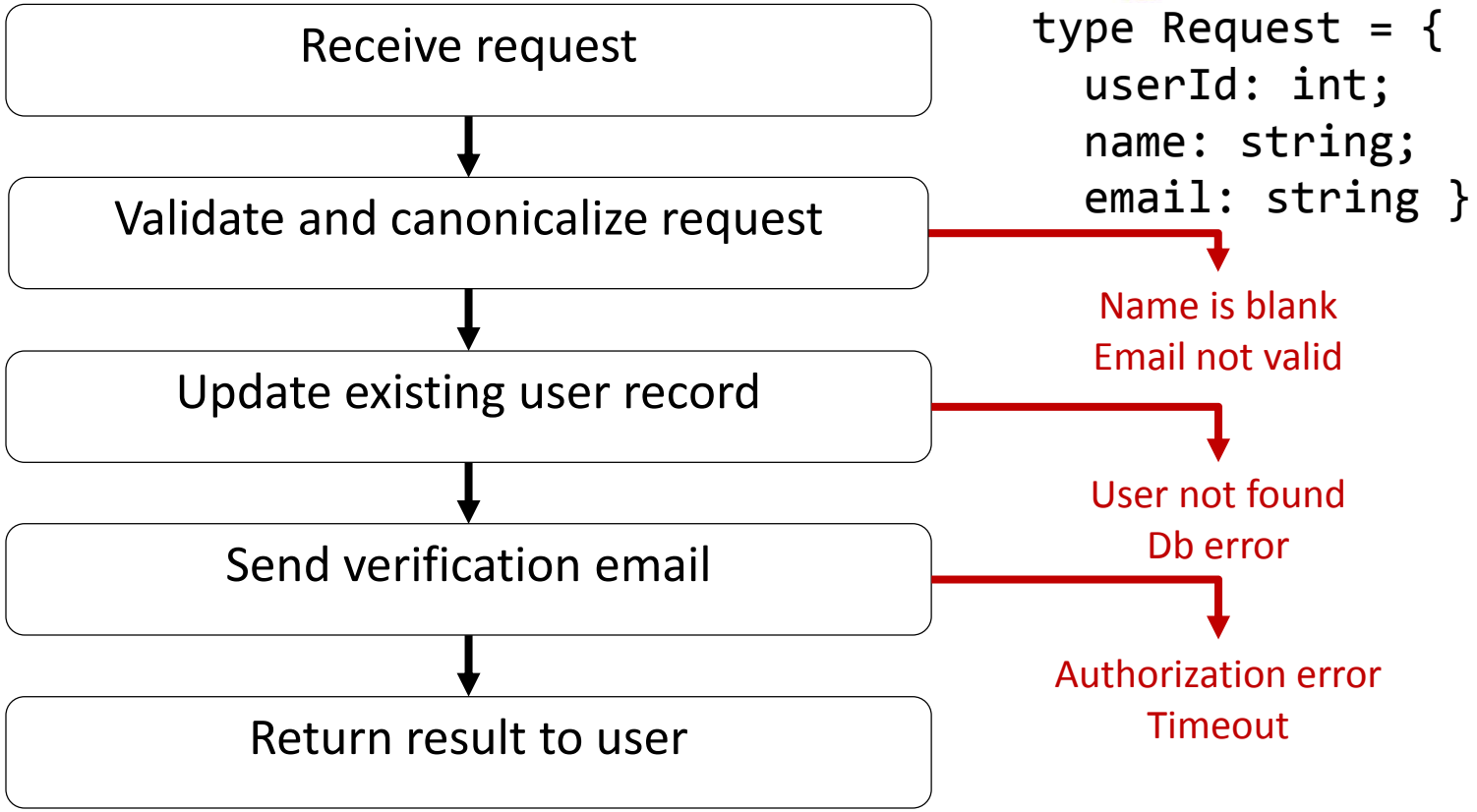

Microsoft Visual Studio
An exception of type 'System.NotImplementedException' occurred in UnhandledExceptionBlog.exe but was not handled in user code
Additional information: The developer needs to do his job.
Form1
Unhandled exception has occurred in your application. If you click Continue, the application will ignore this error and attempt to continue. If you click Quit, the application will close immediately.
ORA-1017: invalid username/password; logon denied.
Details
Continue
Quit
Error
An error has occured while creating an error report
OK
! - Bad User!!!
You've been warned 3 times that this file does not exist.
Now you've made us catch this worthless exception and we're upset.
Do not do this again.
OK
Error
The operation completed successfully.
OK
etails
===== Exception Text ==========
n.IO.IOException: The device is not ready.
ystem.IO.__Error.WinIOError(Int32 errorCod
ystem.IO.FileStream..ctor(String path, FileM
ystem.IO.FileStream..ctor(String path, FileM
rrorHandling.frmErrors.NoErrorHandling()
at ErrorHandling.frmErrors.btnErrorHandling_Clic
at System.Windows.Forms.Control.OnClick(Eve
at System.Windows.Forms.Button.OnClick(Ever
Keyboard not plugged
Windows 95 was unable to detect your keyboard. Press F1 to retry or F2 to abort.
Microsoft Visual Basic
Run-time error '6':
Overflow
Continue
Microsoft Money
An error has ocurred but the error message cannot be retrieved due to another error.
OK
Photosynth Error
Good job – you broke Photosynth!
Okay, it wasn't your fault. We'd love to tell you more, but frankly, we're stumped. It could be caused by something incredibly minor and you'll be able to continue with whatever else you were doing, or maybe Photosynth will crash and burn, perhaps even taking this instance of IE with it. Either way, we're sorry you were inconvenienced.
OK

Straying from the happy path

"As a user I want to update my name and email address"

- and see sensible error messages when something goes wrong!

```
type Request = {
  userId: int;
  name: string;
  email: string }
```



Imperative code with error cases

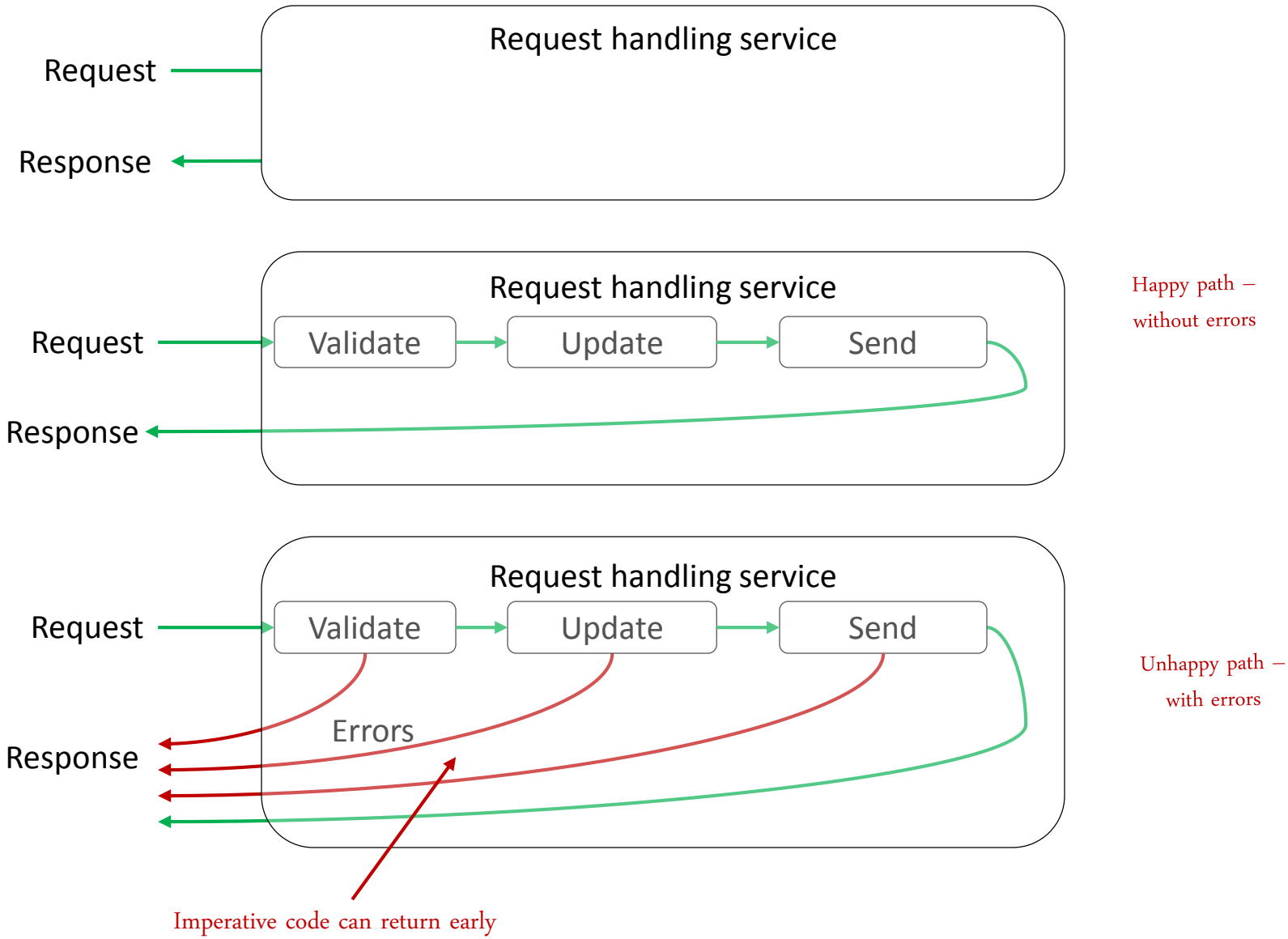
```
string ExecuteUseCase()
{
  var request = receiveRequest();
  var isValidated = validateRequest(request);
  if (!isValidated) {
     return "Request is not valid"
  }
  canonicalizeEmail(request);
  try {
    var result = db.updateDbFromRequest(request);
    if (!result) {
      return "Customer record not found"
    }
  } catch {
    return "DB error: Customer record not updated"
  }

  if (!smtpServer.sendEmail(request.Email)) {
    log.Error "Customer email not sent"
  }

  return "OK";
}
```

Q: What is the functional equivalent of this code?

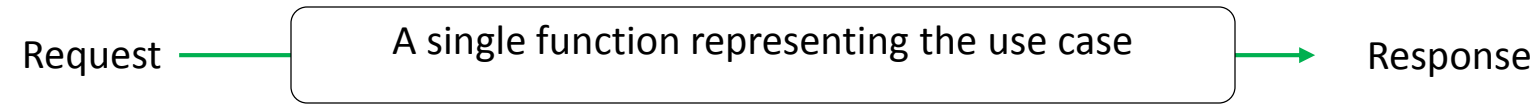
... and can we preserve the elegance of the original functional version?

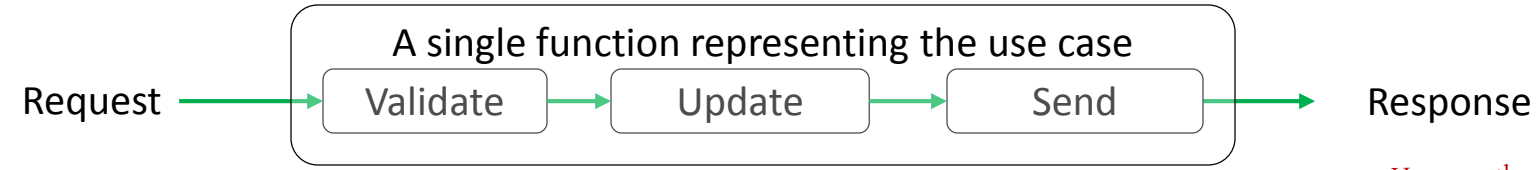
6 clean lines -> 18 ugly lines. 200% extra!
Sadly this is typical of error handling code.

Request/response (non-functional) design

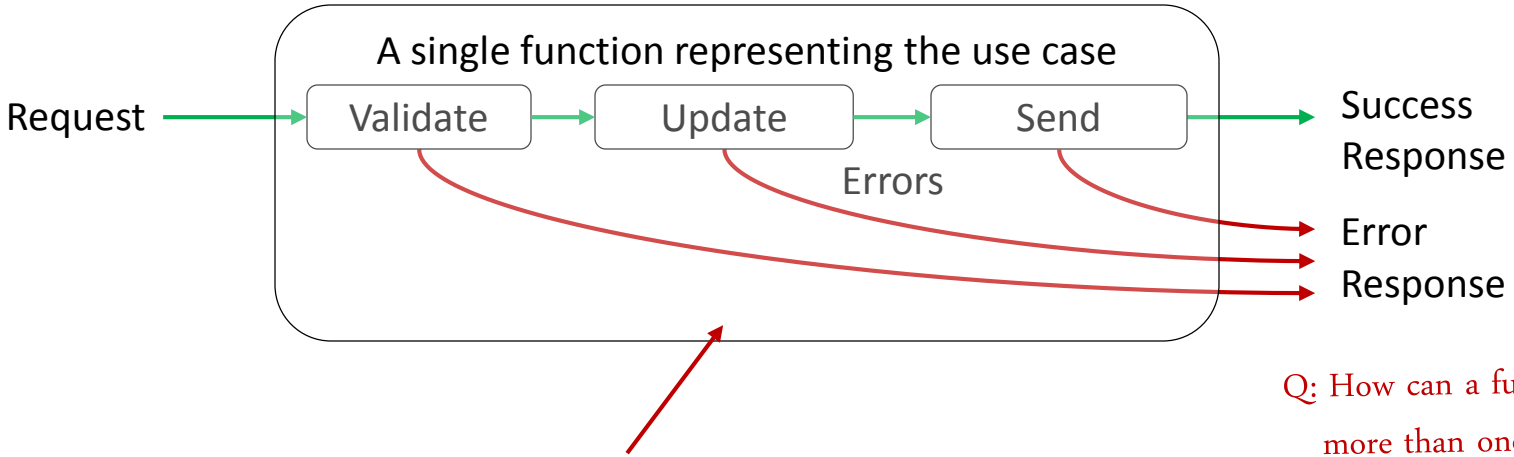


Data flow (functional) design





Happy path – without errors



Q: How can a function have more than one output?

Q: How can you bypass downstream functions when an error happens?

How can a function have more than one output?

```
type Result =
    | Success
    | ValidationError
    | UpdateError
    | SmtpError
```

I love sum types!

But maybe too specific for this case?

How can a function have more than one output?

```
type Result =
    | Success
    | Failure
```

Much more generic – but no data!

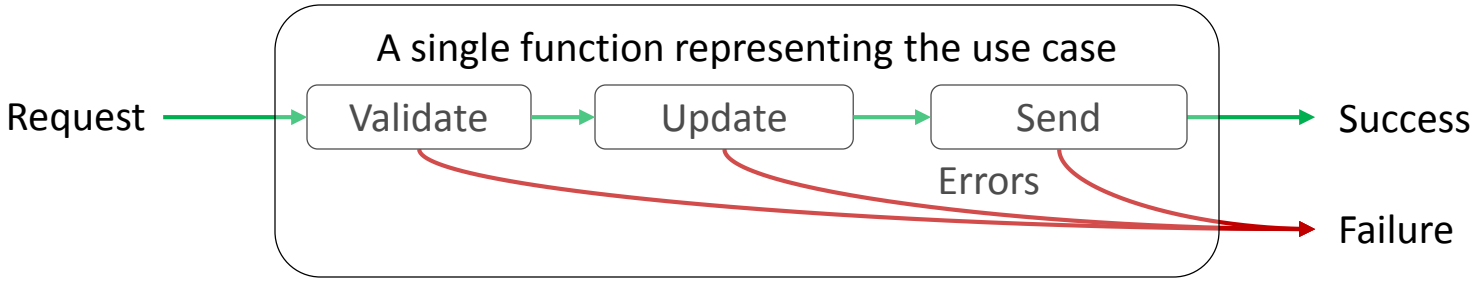
Functional design

How can a function have more than one output?

```
type Result<'TEntity> =
    | Success of 'TEntity
    | Failure of string
```

Good for now – we'll revisit this design later.

Functional design

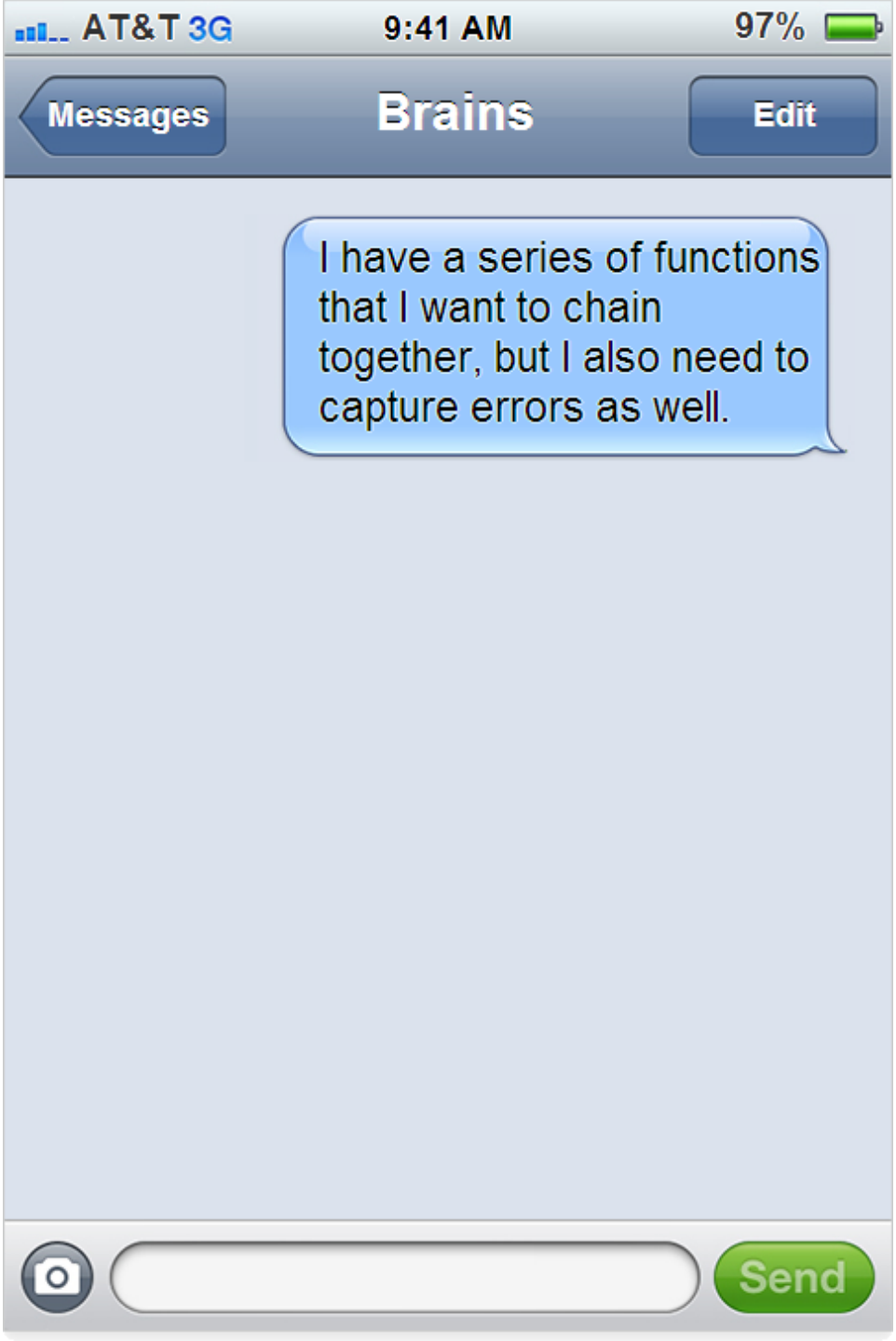


- Each use case will be equivalent to a single function
- The function will return a sum type with two cases:
  "Success" and "Failure".
- The use case function will be built from a series of smaller functions,
  each representing one step in a data flow.
- The errors from each step will be combined into a single "failure" path.

But we haven't answered the question:
How can you bypass downstream functions when an error happens?

How do I work with errors in a functional way?

Very clever



A bear of very little brain

AT&T 3G 9:41 AM 97%

Messages Brains Edit

I have a series of functions that I want to chain together, but I also need to capture errors as well.

That's easy. You just need a monad.

Send

AT&T 3G 9:41 AM 97%

Messages Brains Edit

I have a series of functions that I want to chain together, but I also need to capture errors as well.

That's easy. You just need a monad.

That sounds complicated! What's a monad?

Send

AT&T 3G 9:41 AM 97%

Messages Brains Edit

I have a series of functions that I want to chain together, but I also need to capture errors as well.

That's easy. You just need a monad.

That sounds complicated! What's a monad?

A monad is just a monoid in the category of endofunctors.

Send

AT&T 3G 9:41 AM 97%

Messages Brains Edit

I have a series of functions that I want to chain together, but I also need to capture errors as well.

That's easy. You just need a monad.

That sounds complicated! What's a monad?

A monad is just a monoid in the category of endofunctors.

???

What's the problem?

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

capture errors as well.

That's easy. You just need a monad.

That sounds complicated! What's a monad?

A monad is just a monoid in the category of endofunctors.

???

What's the problem?

I don't know what an endofunctor is :(

Send

AT&T 3G 9:41 AM 97%

Messages Brains Edit

A monad is just a monoid in the category of endofunctors.

???

What's the problem?

I don't know what an endofunctor is :(

That's easy. A functor is a homomorphism between categories, and so an endofunctor is just a functor that maps a category to itself.

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

the category of endofunctors.

???

What's the problem?

I don't know what an endofunctor is :(

That's easy. A functor is a homomorphism between categories, and so an endofunctor is just a functor that maps a category to itself.

Simples!

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

???

What's the problem?

I don't know what an endofunctor is :(

That's easy. A functor is a homomorphism between categories, and so an endofunctor is just a functor that maps a category to itself.

Simples!

Of course! I understand completely now.

Send

AT&T 3G 9:41 AM 97%

Messages Brains Edit

I don't know what an endofunctor is :(

That's easy. A functor is a homomorphism between categories, and so an endofunctor is just a functor that maps a category to itself.

Simples!

Of course! I understand completely now.

But, seriously, what do I have to do?

Send

AT&T 3G 9:41 AM 97%

Messages Brains Edit

homomorphism between categories, and so an endofunctor is just a functor that maps a category to itself.

Simples!

Of course! I understand completely now.

But, seriously, what do I have to do?

Well I suppose you don't really need to know about monads. You only need to use "Maybe".

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

endofunctor is just a functor that maps a category to itself.

Simples!

Of course! I understand completely now.

But, seriously, what do I have to do?

Well I suppose you don't really need to know about monads. You only need to use "Maybe".

Maybe what?

Send

AT&T 3G 9:41 AM 97%

Messages Brains Edit

Simples!

Of course! I understand completely now.

But, seriously, what do I have to do?

Well I suppose you don't really need to know about monads. You only need to use "Maybe".

Maybe what?

"Maybe", the monad.

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

Simples!

Of course! I understand completely now.

But, seriously, what do I have to do?

Well I suppose you don't really need to know about monads. You only need to use "Maybe".

Maybe what?

"Maybe", the monad.

Maybe the monad what?

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

completely now.

But, seriously, what do I have to do?

Well I suppose you don't really need to know about monads. You only need to use "Maybe".

Maybe what?

"Maybe", the monad.

Maybe the monad what?

"Maybe" is the *name* of the monad

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

Well I suppose you don't really need to know about monads. You only need to use "Maybe".

Maybe what?

"Maybe", the monad.

Maybe the monad what?

"Maybe" is the *name* of the monad

Don't you mean, "Maybe the name of the monad is..."

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

Maybe what?

"Maybe", the monad.

Maybe the monad what?

"Maybe" is the *name* of the monad

Don't you mean, "Maybe the name of the monad is..."

"Maybe" the name of the monad is? You're talking just like Yoda.

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

"Maybe", the monad.

Maybe the monad what?

"Maybe" is the *name* of the monad

Don't you mean, "Maybe the name of the monad is..."

"Maybe" the name of the monad is? You're talking just like Yoda.

No, talking like Yoda you are!

Send

AT&T 3G 9:41 AM 97%

Messages Brains Edit

"Maybe" is the *name* of the monad

Don't you mean, "Maybe the name of the monad is..."

"Maybe" the name of the monad is? You're talking just like Yoda.

No, talking like Yoda you are!

Getting back on topic... "Maybe" is certainly what you want.

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

monad

Don't you mean, "Maybe the name of the monad is..."

"Maybe" the name of the monad is? You're talking just like Yoda.

No, talking like Yoda you are!

Getting back on topic... "Maybe" is certainly what you want.

Definitely Maybe, then?

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

the name of the monad is..."

"Maybe" the name of the monad is? You're talking just like Yoda.

No, talking like Yoda you are!

Getting back on topic... "Maybe" is certainly what you want.

Definitely Maybe, then?

Actually, I prefer "(What's the Story) Morning Glory?"

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

"Maybe" the name of the monad is? You're talking just like Yoda.

No, talking like Yoda you are!

Getting back on topic... "Maybe" is certainly what you want.

Definitely Maybe, then?

Actually, I prefer "(What's the Story) Morning Glory?"

Now I think about it, "Either" might be better...

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

like Yoda.

No, talking like Yoda you are!

Getting back on topic... "Maybe" is certainly what you want.

Definitely Maybe, then?

Actually, I prefer "(What's the Story) Morning Glory?"

Now I think about it, "Either" might be better...

Either what?

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

No, talking like Yoda you are!

Getting back on topic... "Maybe" is certainly what you want.

Definitely Maybe, then?

Actually, I prefer "(What's the Story) Morning Glory?"

Now I think about it, "Either" might be better...

Either what?

"Either", the monad

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

Getting back on topic... "Maybe" is certainly what you want.

Definitely Maybe, then?

Actually, I prefer "(What's the Story) Morning Glory?"

Now I think about it, "Either" might be better...

Either what?

"Either", the monad

Either the monad or what?

Send

AT&T 3G 9:41 AM 97%

Messages Brains Edit

you want.

Definitely Maybe, then?

Actually, I prefer "(What's the Story) Morning Glory?"

Now I think about it, "Either" might be better...

Either what?

"Either", the monad

Either the monad or what?

"Either", that's all.

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

Definitely Maybe, then?

Actually, I prefer "(What's the Story) Morning Glory?"

Now I think about it, "Either" might be better...

Either what?

"Either", the monad

Either the monad or what?

"Either", that's all.

Just Either?

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

the Story) Morning Glory?"

Now I think about it, "Either" might be better...

Either what?

"Either", the monad

Either the monad or what?

"Either", that's all.

Just Either?

No, no, no! Just is part of Maybe

Send

AT&T 3G 9:41 AM 97%

Messages Brains Edit

Now I think about it, "Either" might be better...

Either what?

"Either", the monad

Either the monad or what?

"Either", that's all.

Just Either?

No, no, no! Just is part of Maybe

Just Maybe?

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

Either what?

"Either", the monad

Either the monad or what?

"Either", that's all.

Just Either?

No, no, no! Just is part of Maybe

Just Maybe?

No, you have to say Just Just or Just Nothing

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

Either the monad or what?

"Either", that's all.

Just Either?

No, no, no! Just is part of Maybe

Just Maybe?

No, you have to say Just Just or Just Nothing

Just Nothing? But a minute ago you said Definitely Maybe!

Send

AT&T 3G 9:41 AM 97%

Messages Brains Edit

Either", that's all.

Just Either?

No, no, no! Just is part of Maybe

Just Maybe?

No, you have to say Just Just or Just Nothing

Just Nothing? But a minute ago you said Definitely Maybe!

Well I'm talking about "Either" now

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

Maybe

Just Maybe?

No, you have to say Just Just or Just Nothing

Just Nothing? But a minute ago you said Definitely Maybe!

Well I'm talking about "Either" now

Either "Just Nothing" or "Definitely Maybe" -- which one is it? Make up your mind!

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

No, you have to say Just Just or Just Nothing

Just Nothing? But a minute ago you said Definitely Maybe!

Well I'm talking about "Either" now

Either "Just Nothing" or "Definitely Maybe" -- which one is it? Make up your mind!

Neither! I told you, you should use Either.

Send

AT&T 3G 9:41 AM 97%

Messages **Brains** Edit

Just or Just Nothing

Just Nothing? But a minute ago you said Definitely Maybe!

Well I'm talking about "Either" now

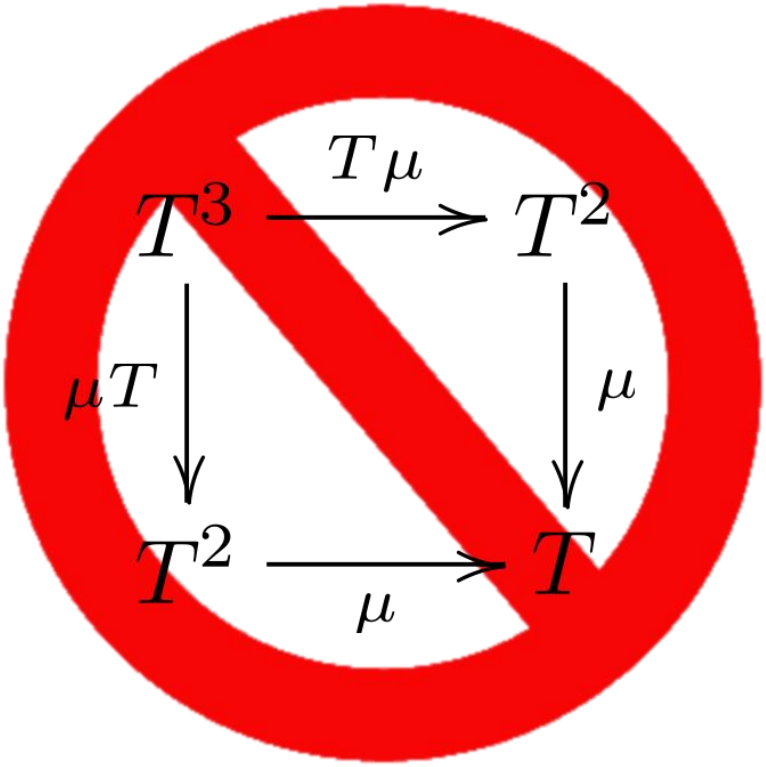
Either "Just Nothing" or "Definitely Maybe" -- which one is it? Make up your mind!

Neither! I told you, you should use Either.

Aaargh!

Send

Monads are confusing

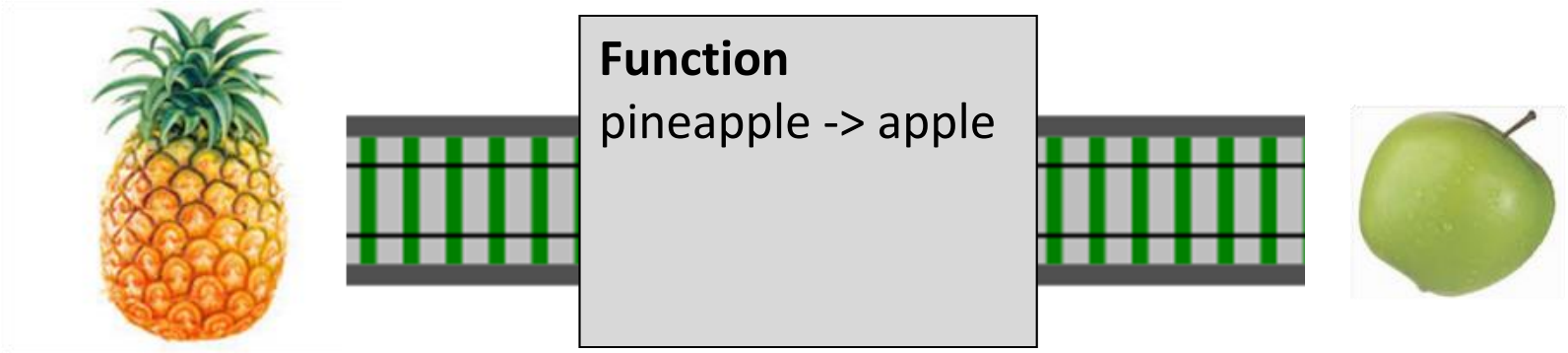


Monad Free Zone

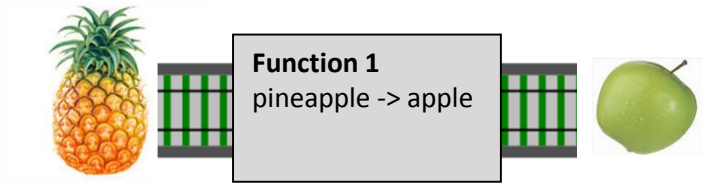
Railway oriented programming

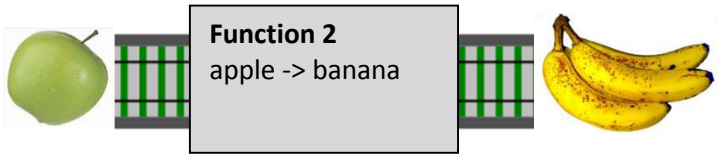
This has absolutely nothing to do with monads.

A railway track analogy

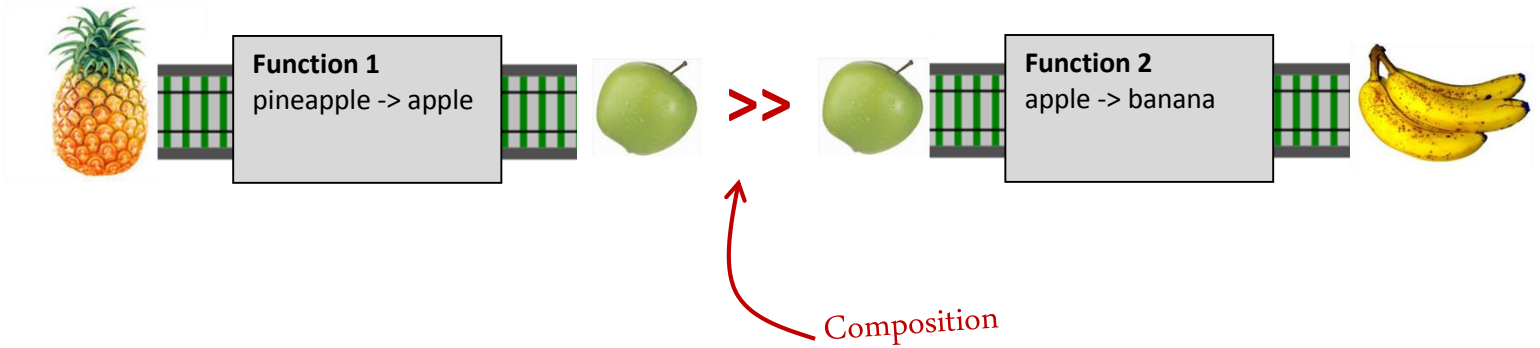


A railway track analogy

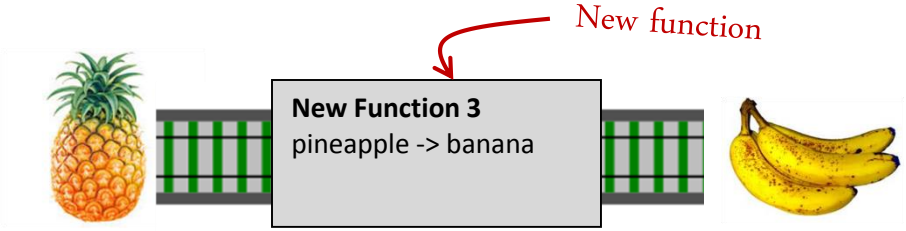




A railway track analogy



A railway track analogy



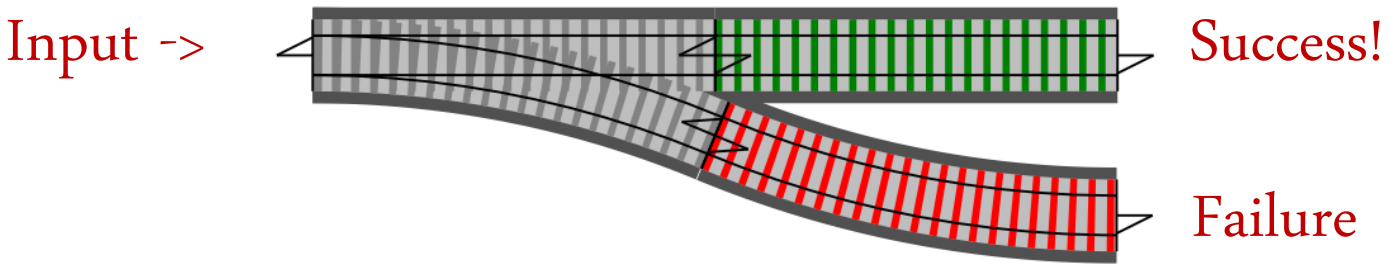
Can't tell it was built from smaller functions!

An error generating function

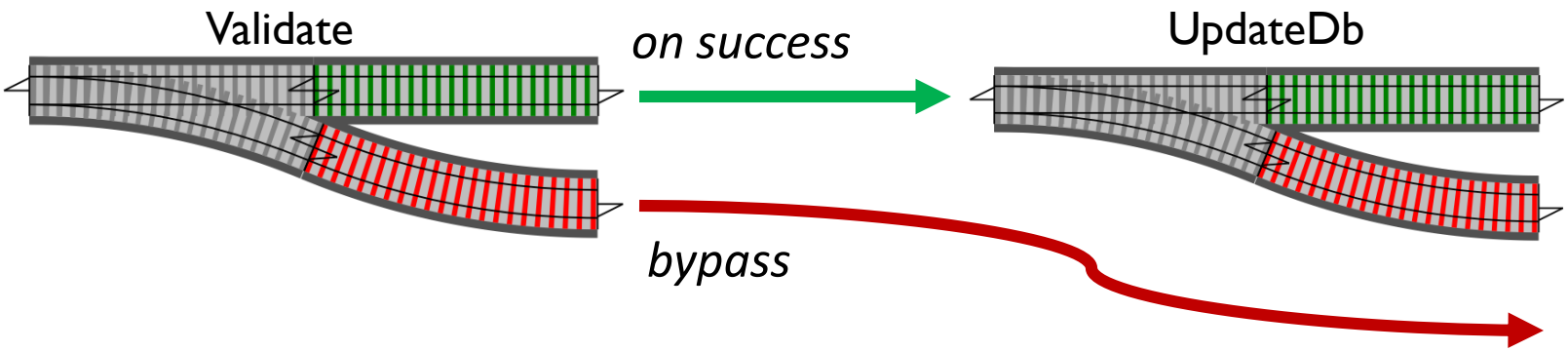
How do we model this as railway track?

```
let validateInput input =
    if input.name = "" then
        Failure "Name must not be blank"
    else if input.email = "" then
        Failure "Email must not be blank"
    else
        Success input  // happy path
```

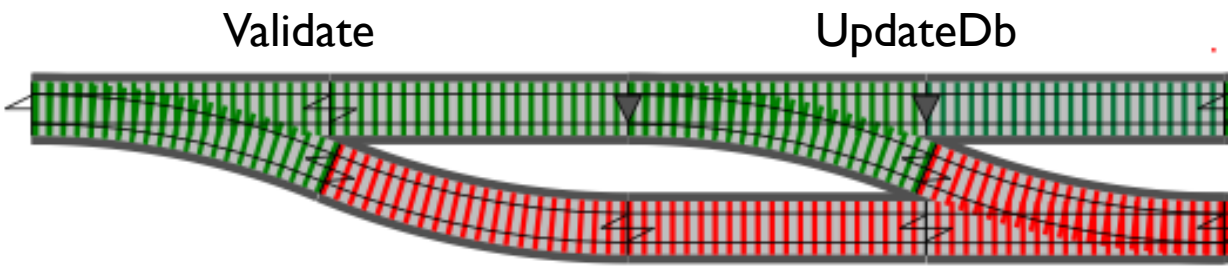
Introducing switches



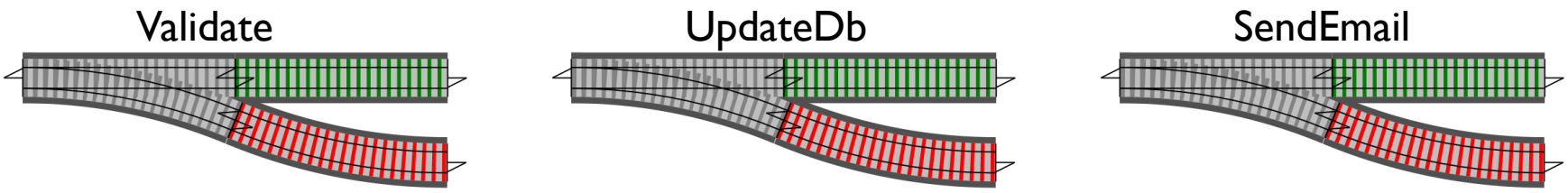
Connecting switches



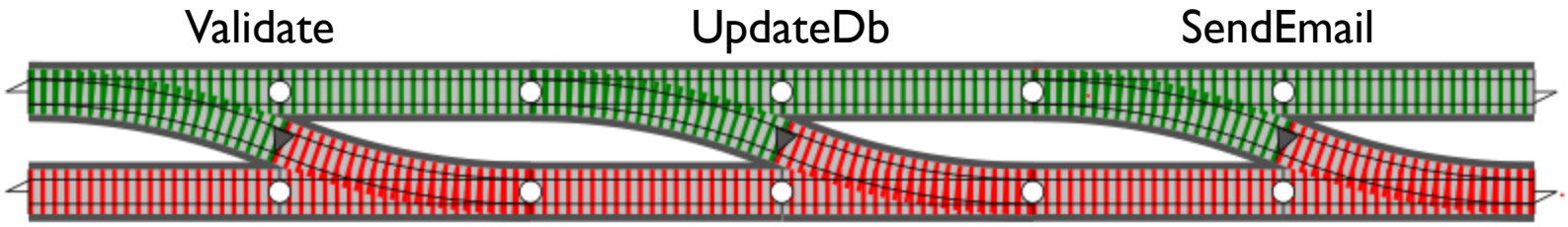
Connecting switches



Connecting switches



Connecting switches



This is the "two track" model –

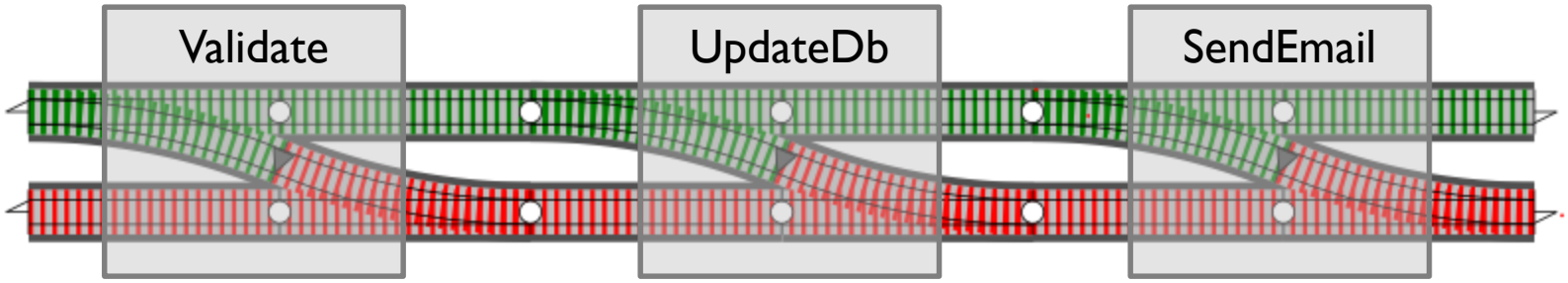
the basis for the "Railway Oriented Programming" approach to error handling.

The two-track model in practice

Composing switches

Here we have a series of black box functions that are straddling a two-track railway.

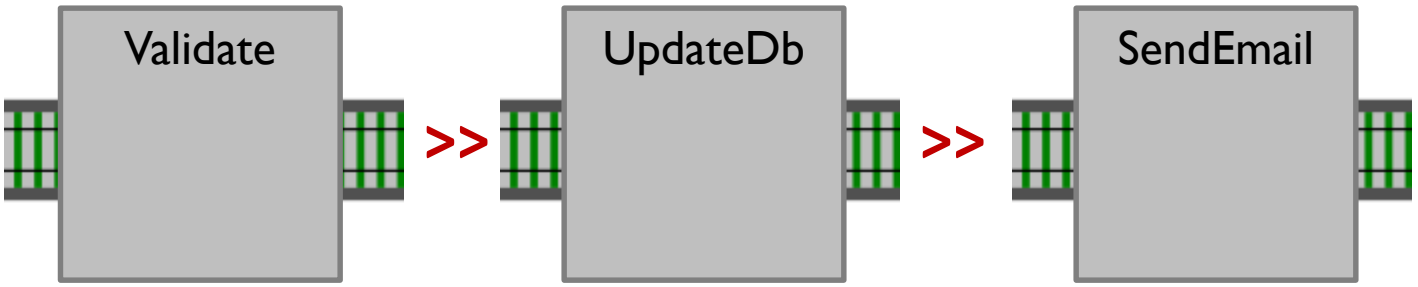
Composing switches



Here we have a series of black box functions that are straddling a two-track railway.

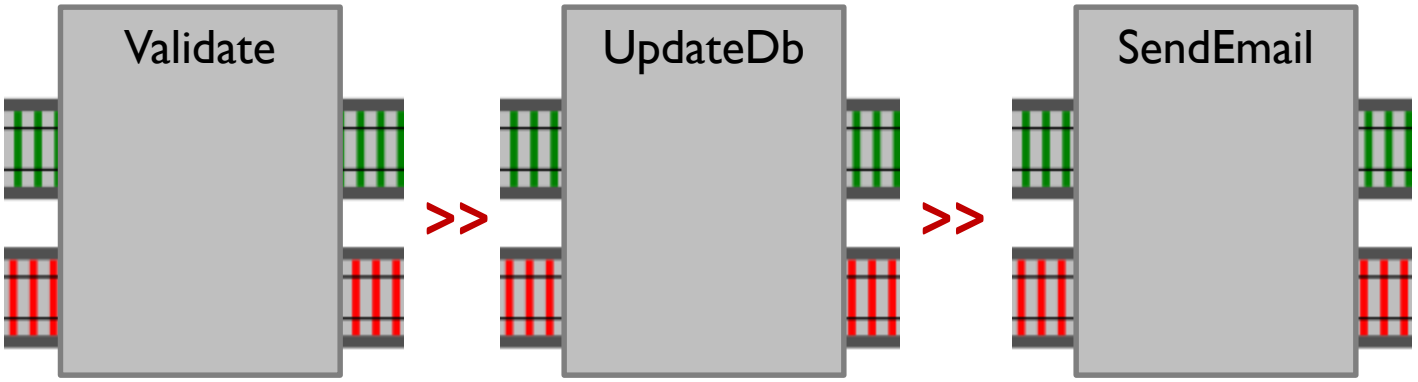
Inside each box there is a switch function.

Composing switches



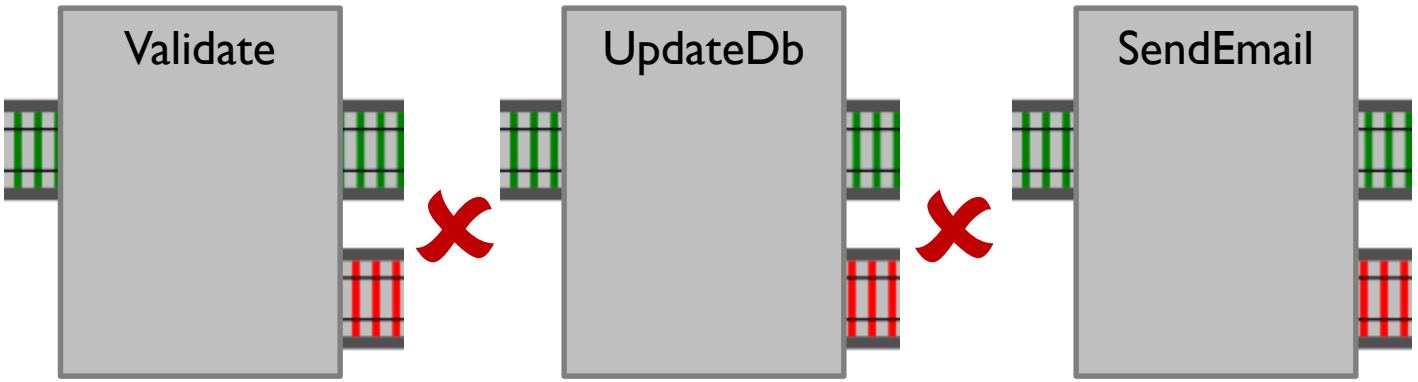
Composing one-track functions is fine...

Composing switches



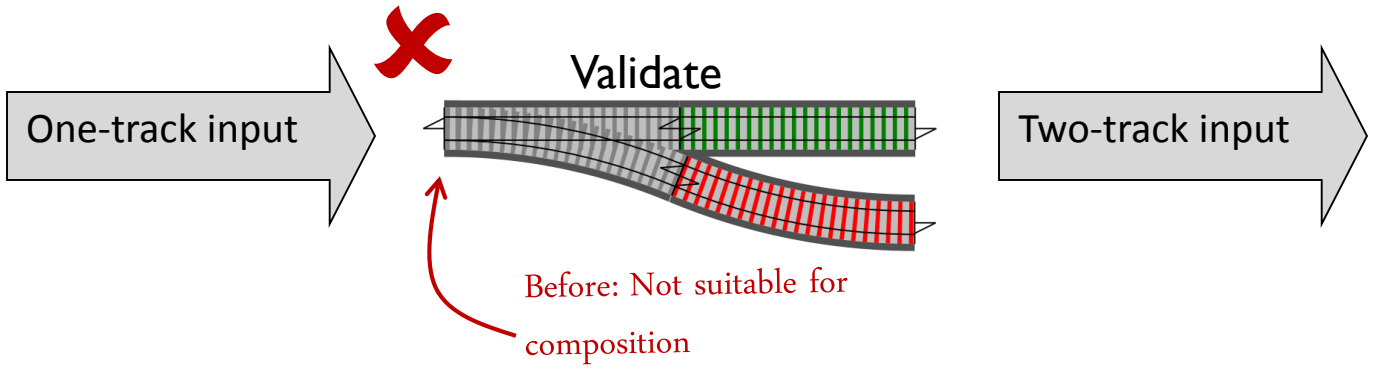
... and composing two-track functions is fine...

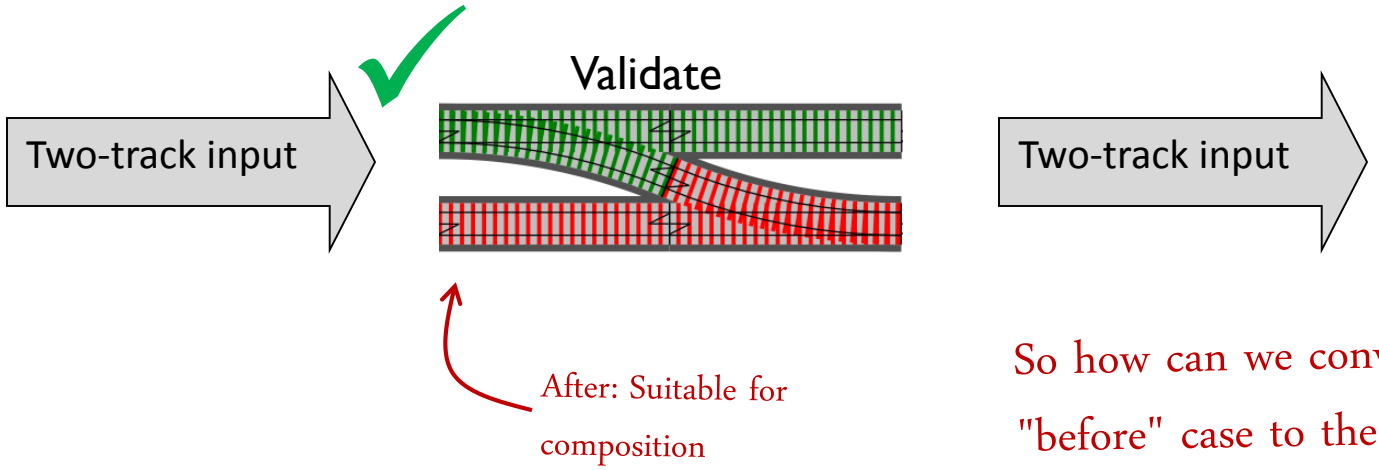
Composing switches



... but composing switches is not allowed!

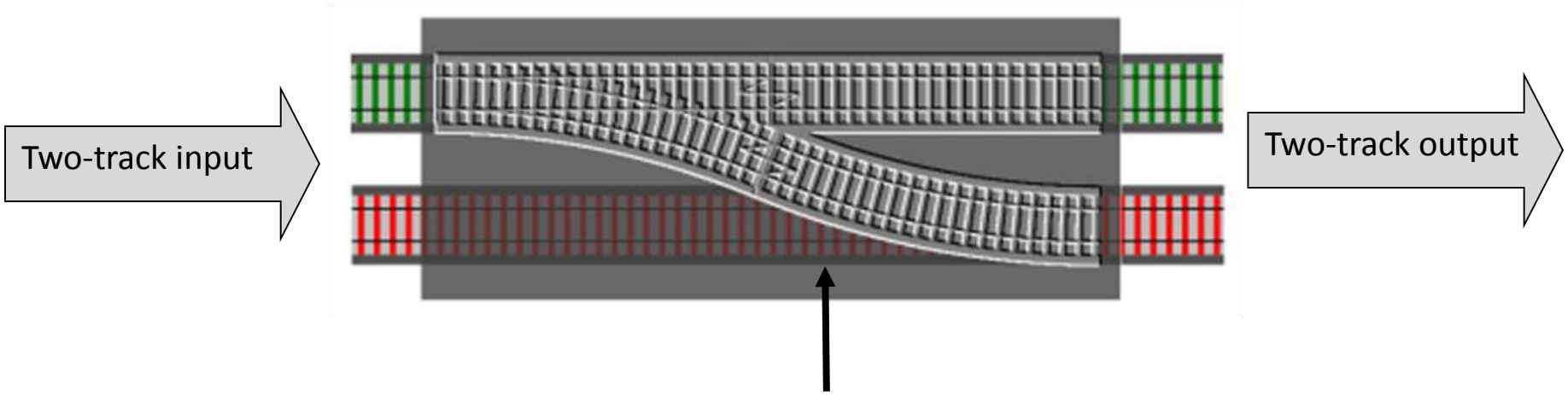
Composing switches



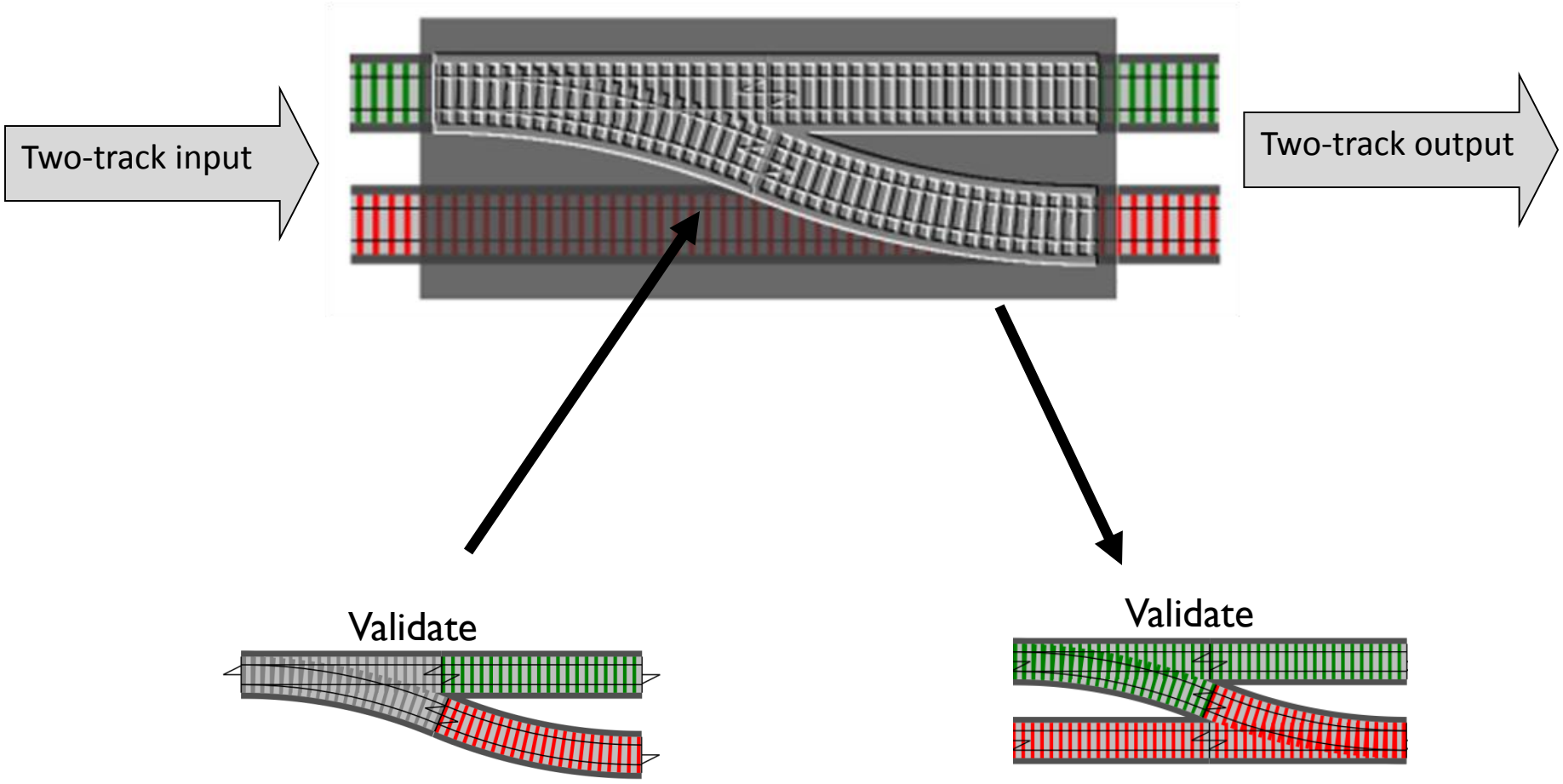


So how can we convert from the "before" case to the "after" case?

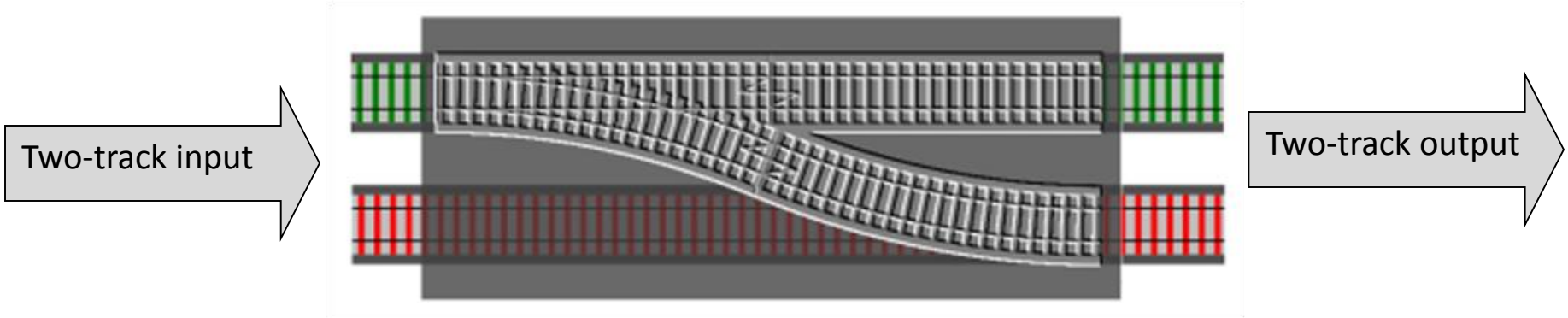
Bind as an adapter block



Bind as an adapter block



Bind as an adapter block




```
let bind switchFunction =
    fun twoTrackInput ->
        match twoTrackInput with
        | Success s -> switchFunction s
        | Failure f -> Failure f
```

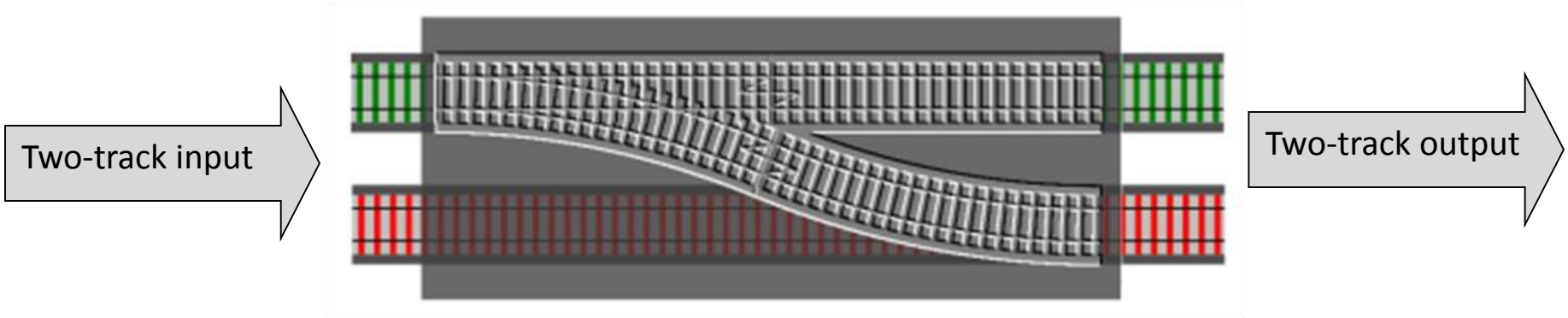
```
bind : ('a -> Result<'b>) -> Result<'a> -> Result<'b>
```

Switch function

2-track input

2-track output

Bind as an adapter block




```
let bind switchFunction twoTrackInput =
    match twoTrackInput with
    | Success s -> switchFunction s
    | Failure f -> Failure f
```

Same function: alternative version with two parameters.

```
bind : ('a -> Result<'b>) -> Result<'a> -> Result<'b>
```

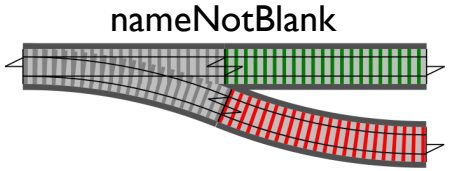
Switch function

2-track input

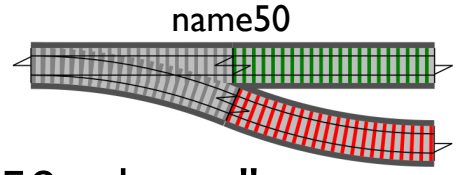
2-track output

Bind example

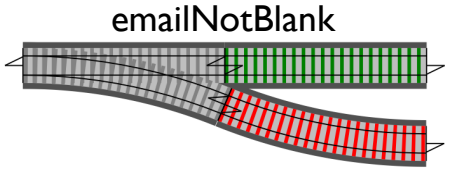
```
let nameNotBlank input =
  if input.name = "" then
     Failure "Name must not be blank"
  else Success input
```



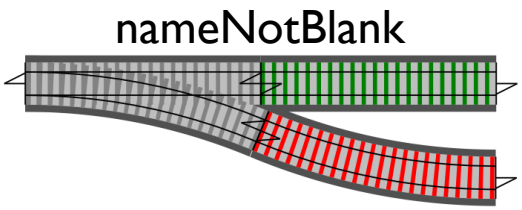

```
let name50 input =
  if input.name.Length > 50 then
     Failure "Name must not be longer than 50 chars"
  else Success input
```

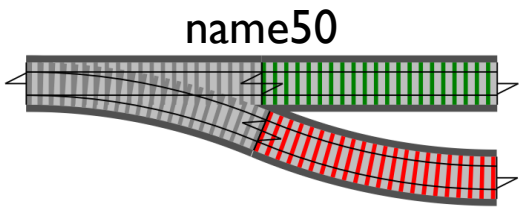


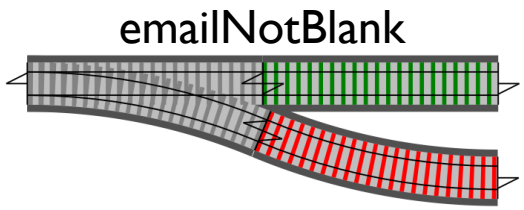

```
let emailNotBlank input =
  if input.email = "" then
     Failure "Email must not be blank"
  else Success input
```



Bind example

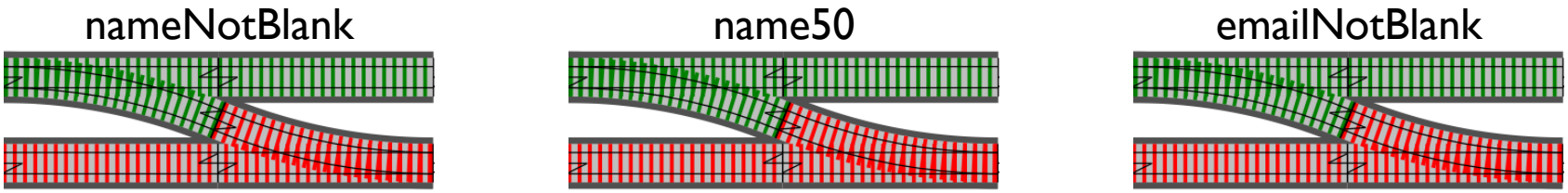







```
nameNotBlank (combined with)
name50 (combined with)
emailNotBlank
```

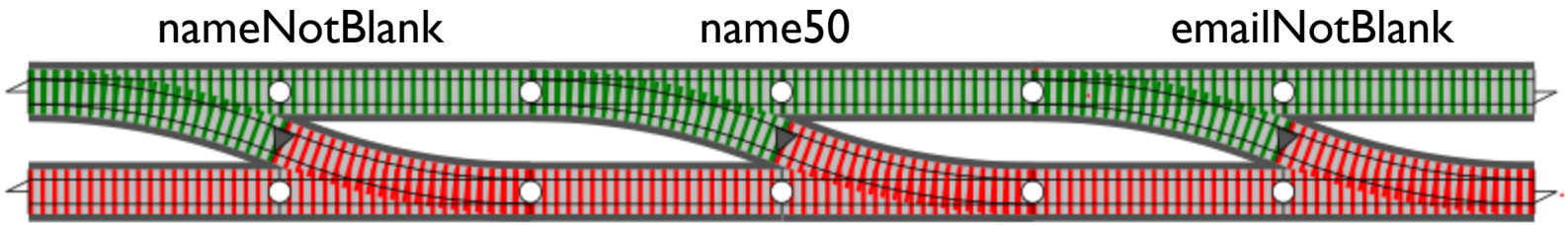
Bind example




```
bind nameNotBlank
bind name50
bind emailNotBlank
```

use "bind" to convert to 2-track

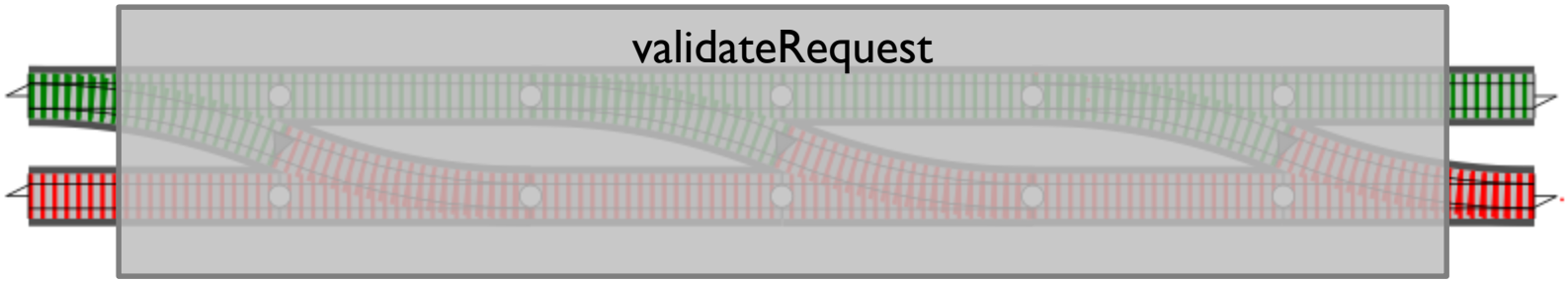
Bind example




```
bind nameNotBlank
>> bind name50
>> bind emailNotBlank
```

then compose together

Bind example



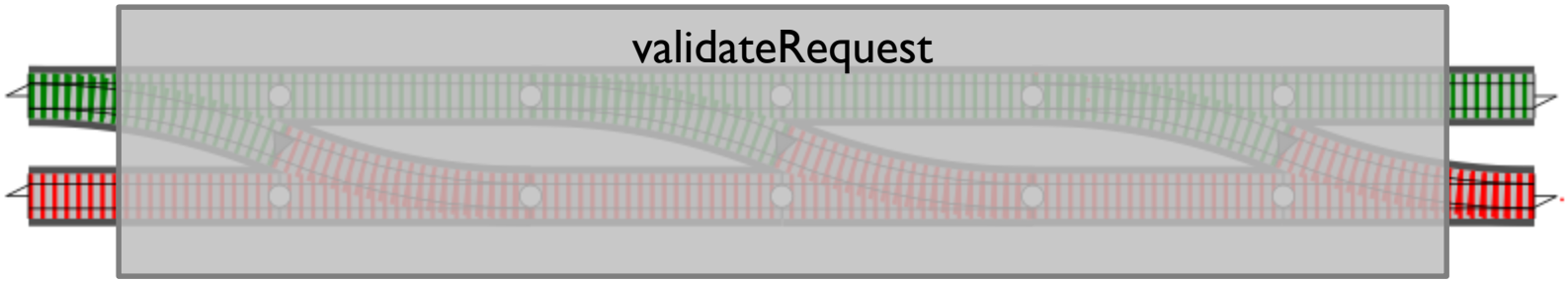
Define a function

```
let validateRequest =
  bind nameNotBlank
  >> bind name50
  >> bind emailNotBlank
```

```
// validateRequest : Result<Request> -> Result<Request>
```

Overall result is a new two-track function

Bind example



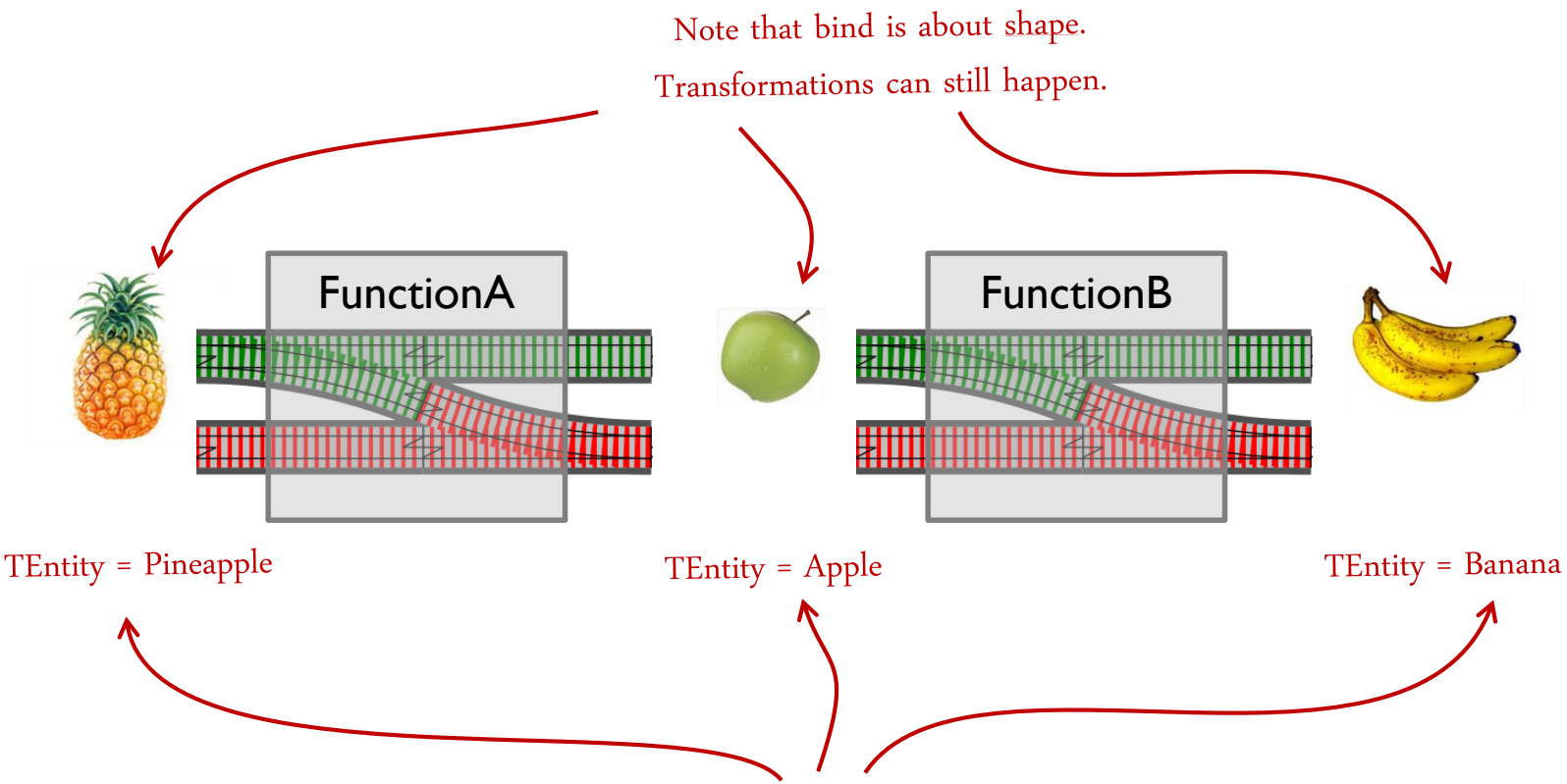
Common symbol for bind

```
let (>>=) twoTrackInput switchFunction =
  bind switchFunction twoTrackInput

let validateRequest twoTrackInput =
  twoTrackInput
  >>= nameNotBlank
  >>= name50
  >>= emailNotBlank
```

Needs a explicit parameter

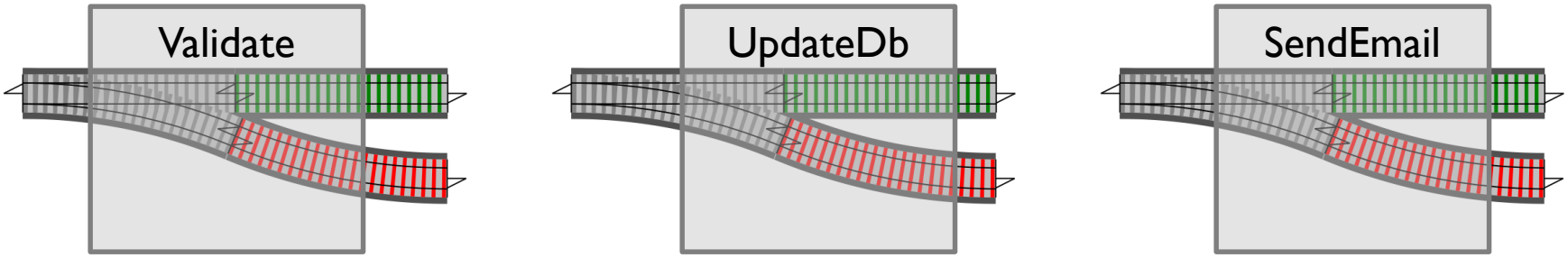
Bind symbol = F# composition symbol + railway track symbol! Coincidence?

Bind doesn't stop transformations

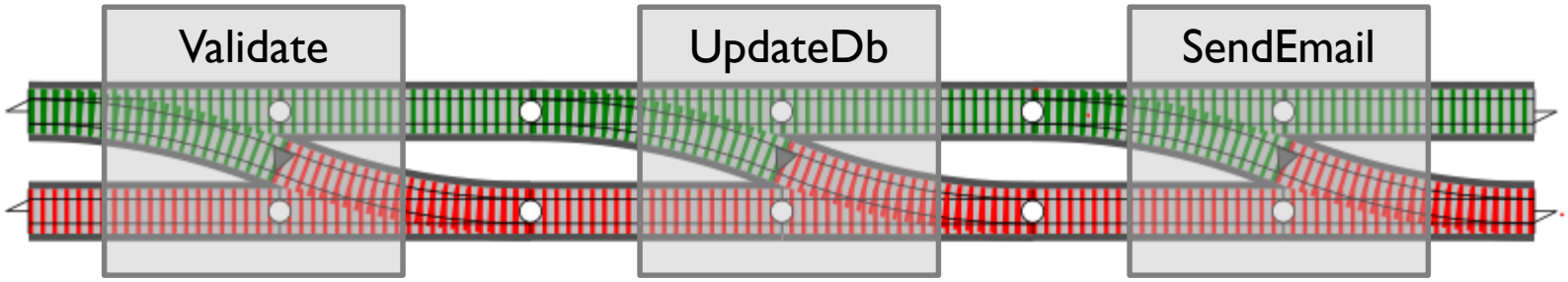



```
type Result<'TEntity> =
    | Success of 'TEntity
    | Failure of string
```

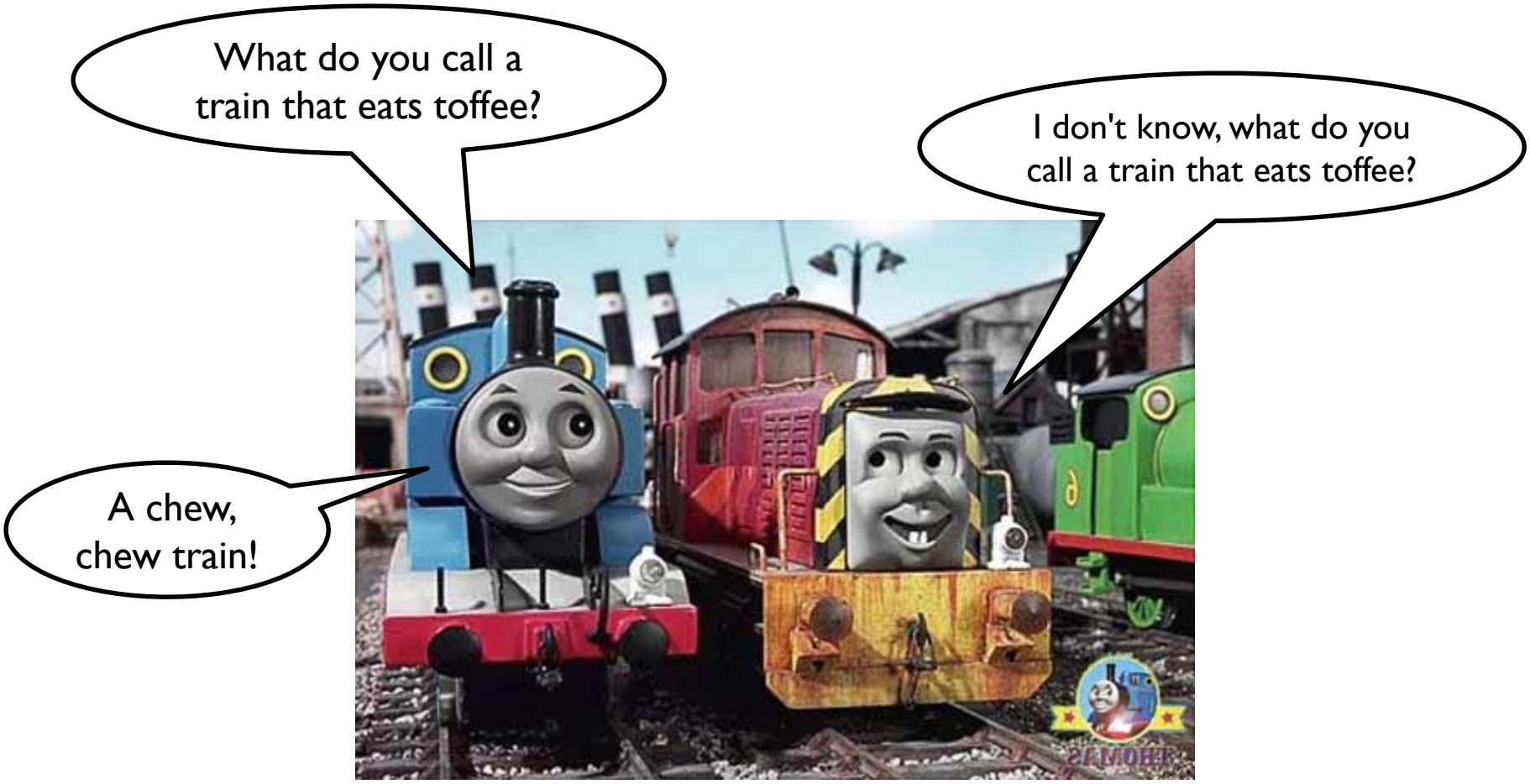
Composing switches - review



Converted to two-track functions using bind



Comic Interlude



More fun with railway tracks...

...extending the framework

More fun with railway tracks...

Fitting other functions into this framework:

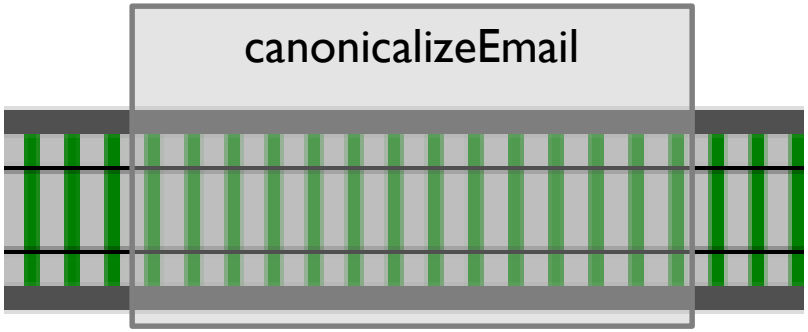
- Single track functions
- Dead-end functions
- Functions that throw exceptions
- Supervisory functions

Converting one-track functions

Fitting other functions into this framework:

- **Single track functions**
- Dead-end functions
- Functions that throw exceptions
- Supervisory functions

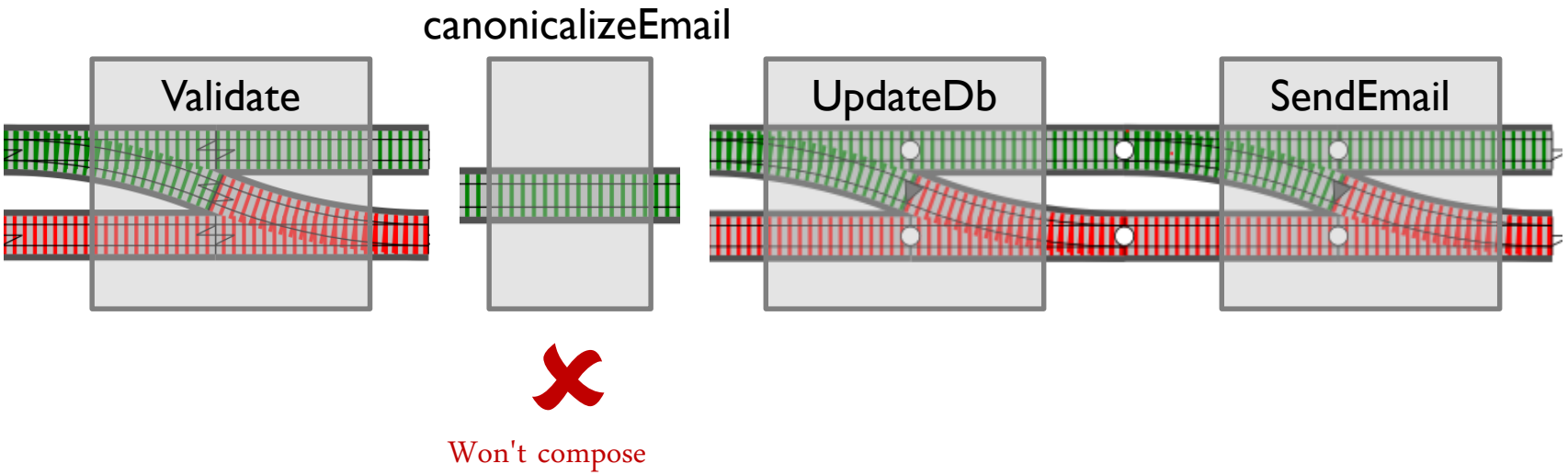
Converting one-track functions



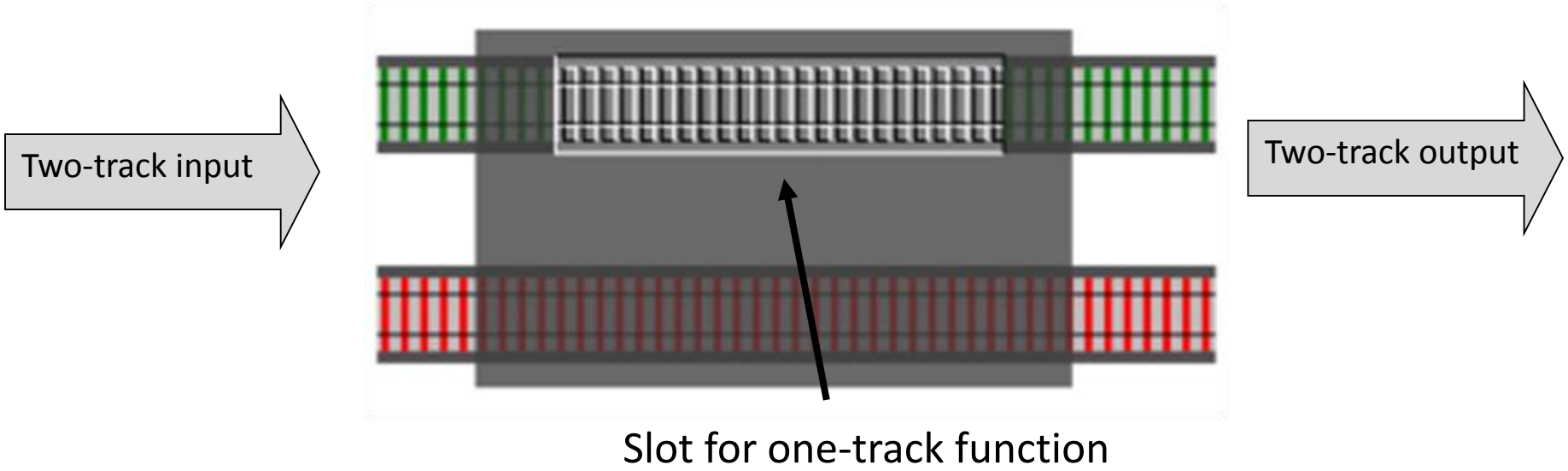

```
// trim spaces and lowercase
let canonicalizeEmail input =
   { input with email = input.email.Trim().ToLower() }
```

A simple function that doesn't generate errors – a "one-track" function.

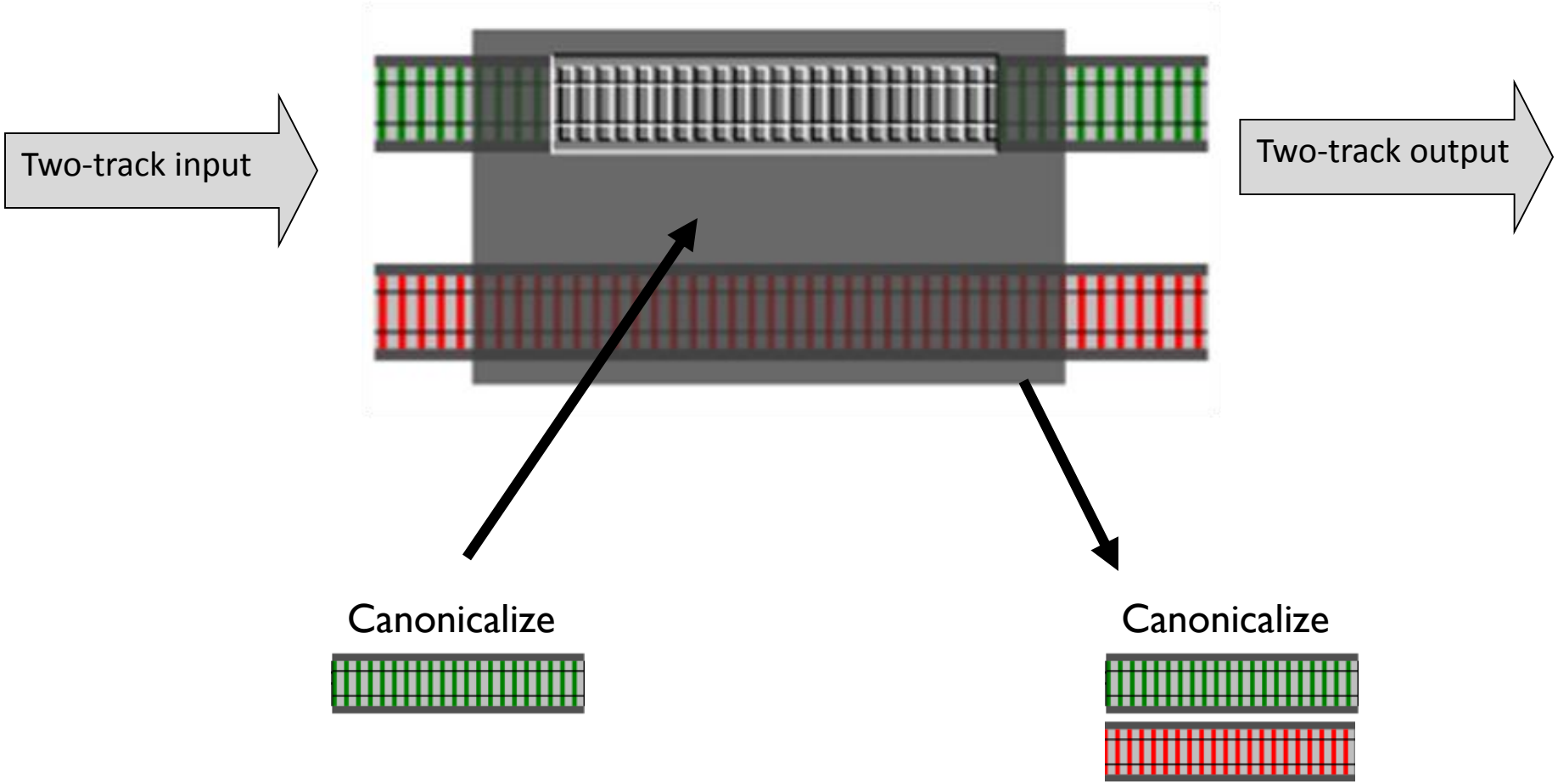
Converting one-track functions



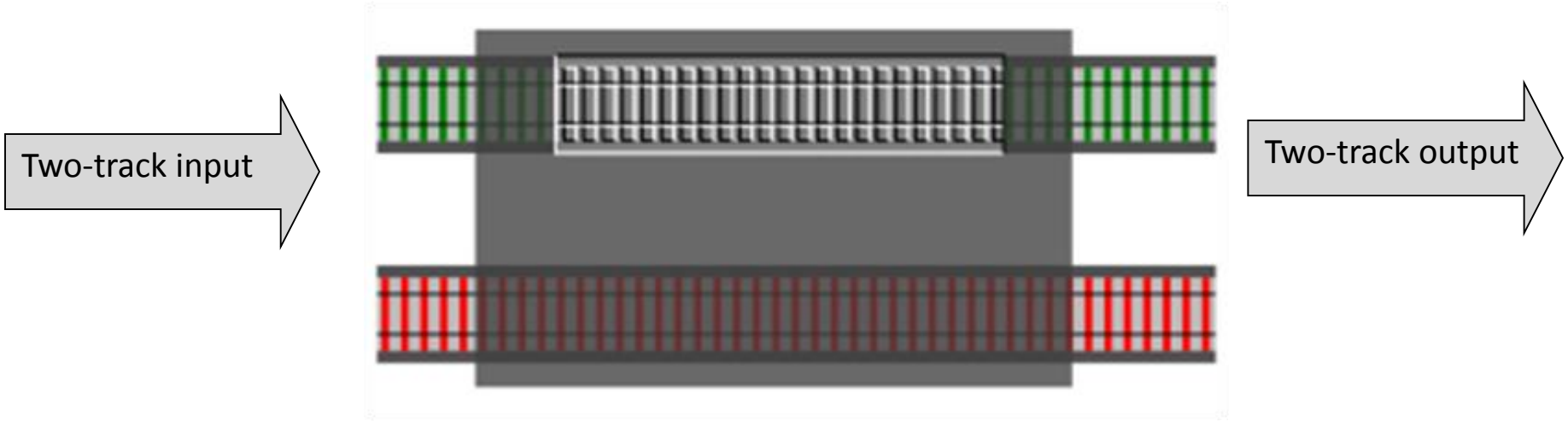
Converting one-track functions



Converting one-track functions



Converting one-track functions




```
let map singleTrackFunction twoTrackInput =
  match twoTrackInput with
  | Success s -> Success (singleTrackFunction s)
  | Failure f -> Failure f
```

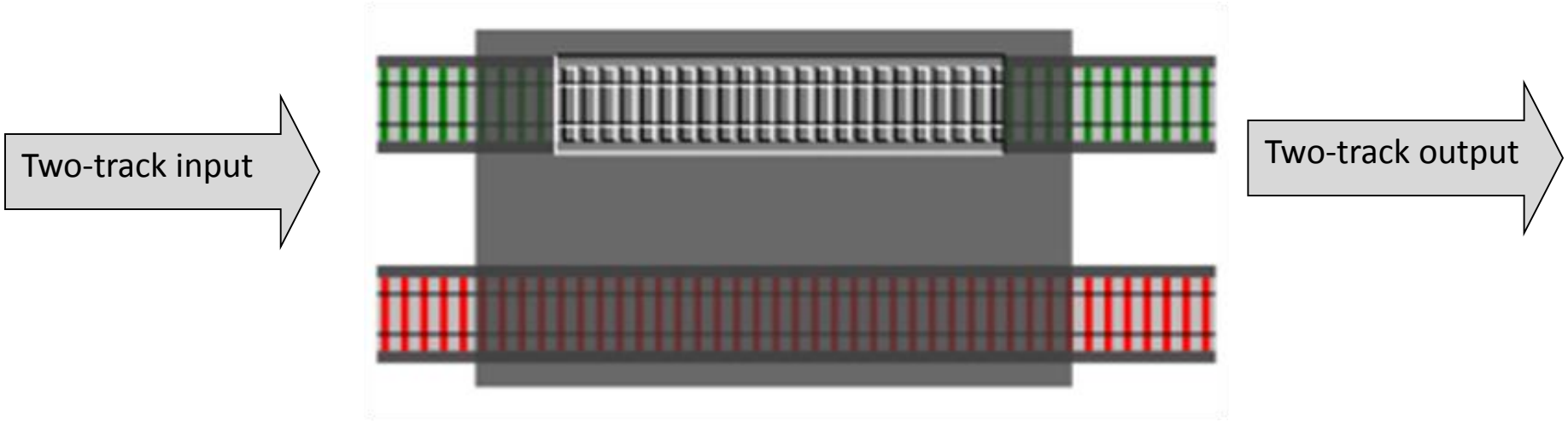
```
map : ('a -> 'b) -> Result<'a> -> Result<'b>
```

Single track function

2-track input

2-track output

Converting one-track functions




```
let map singleTrackFunction =
  bind (singleTrackFunction >> Success)
```

Tip: "map" can also be built from "bind" and "Success"

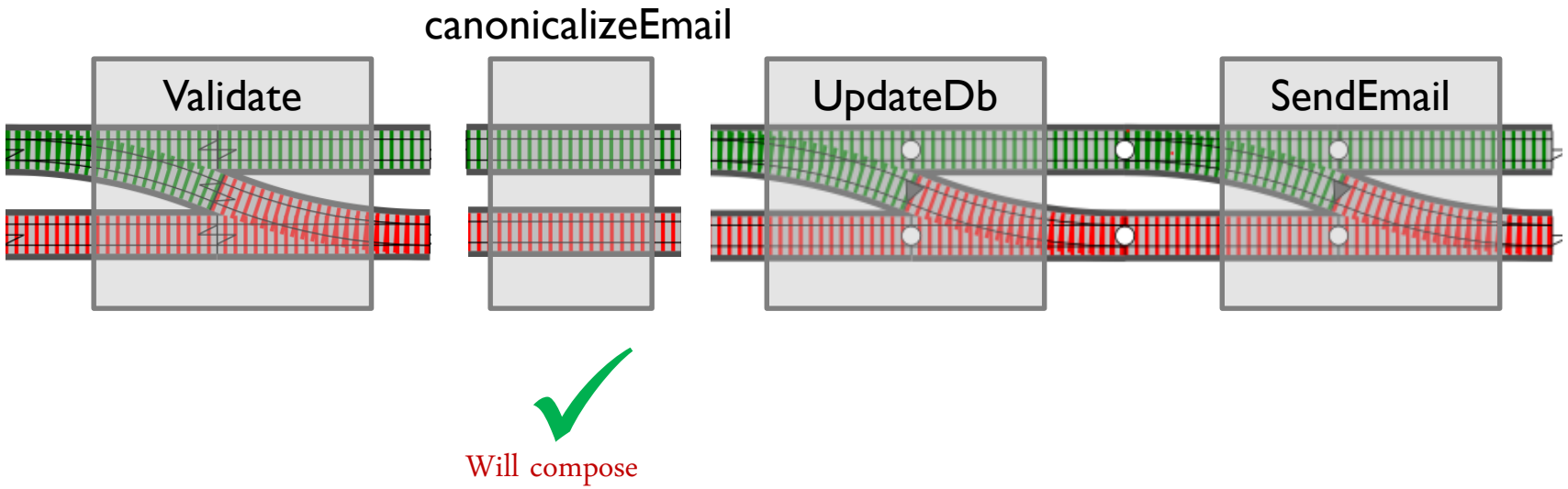
```
map : ('a -> 'b) -> Result<'a> -> Result<'b>
```

Single track function

2-track input

2-track output

Converting one-track functions

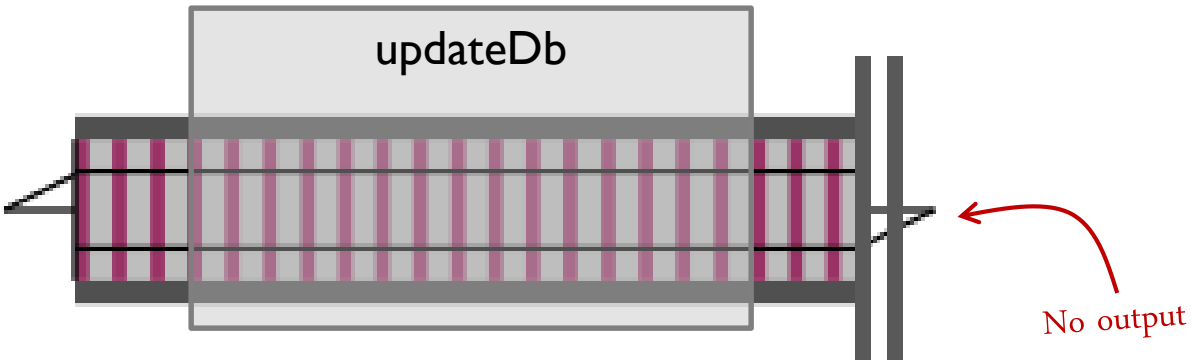


Converting dead-end functions

Fitting other functions into this framework:

- Single track functions
- **Dead-end functions**
- Functions that throw exceptions
- Supervisory functions

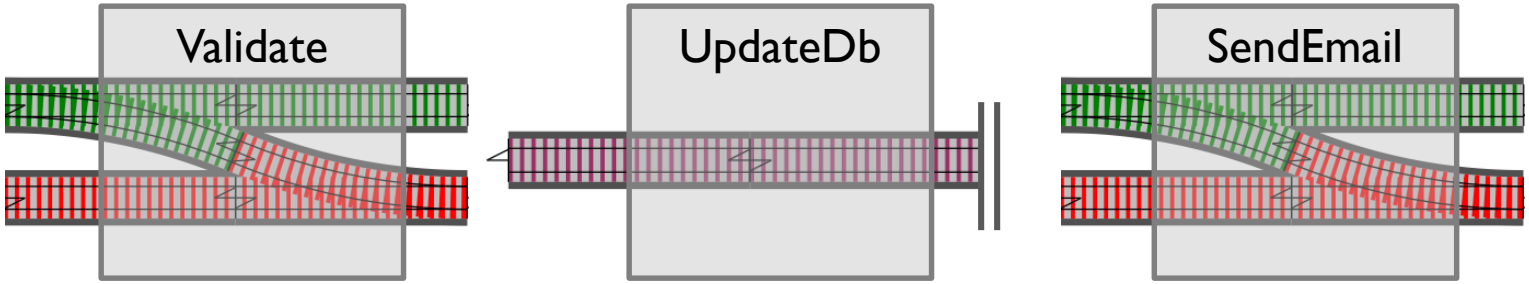
Converting dead-end functions




```
let updateDb request =
    // do something
    // return nothing at all
```

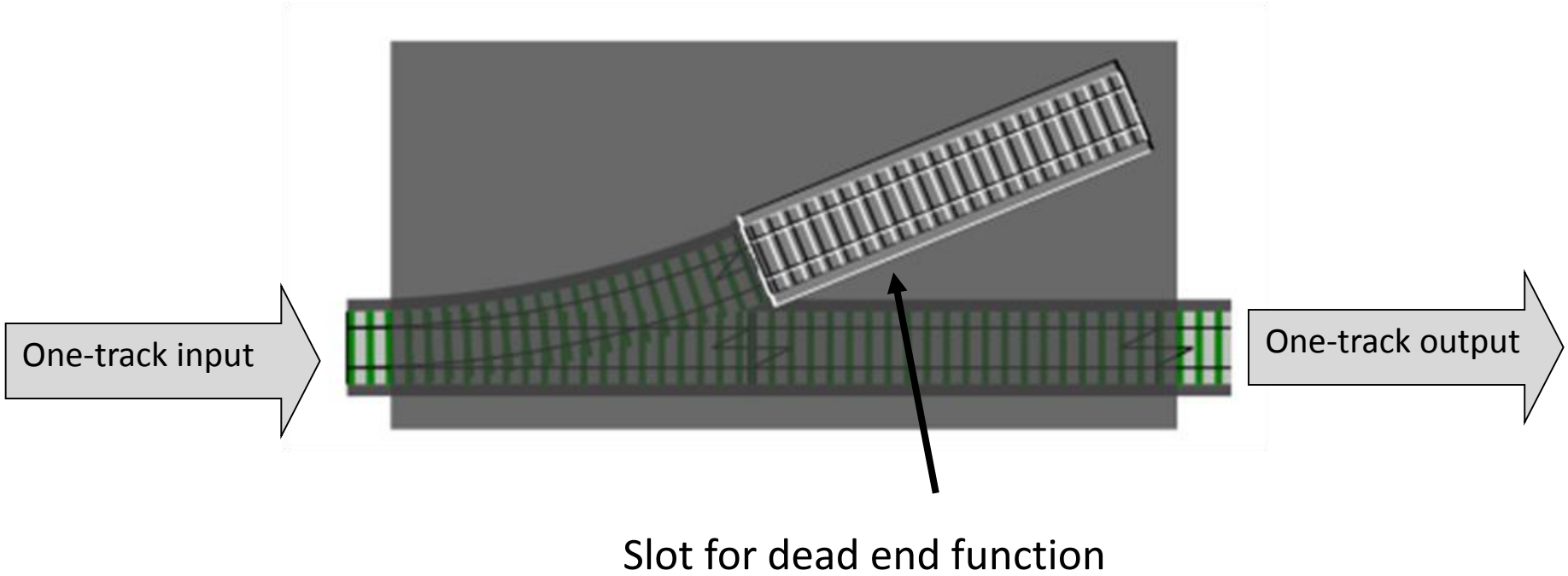
A function that doesn't return anything– a "dead-end" function.

Converting dead-end functions

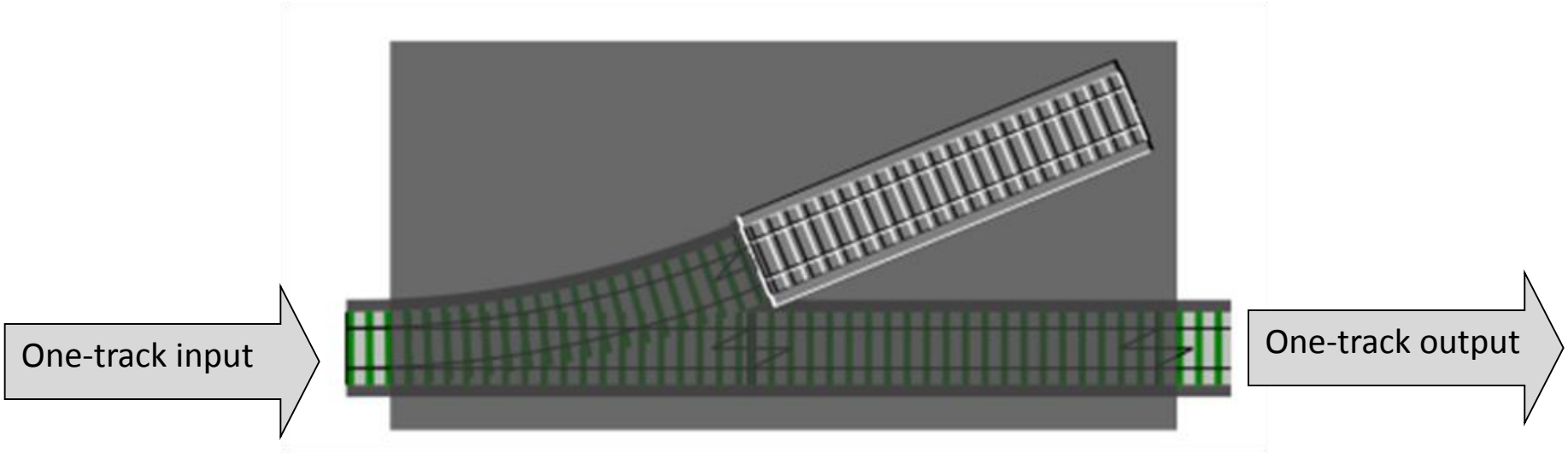


Won't compose

Converting dead-end functions



Converting dead-end functions




```
let tee deadEndFunction oneTrackInput =
    deadEndFunction oneTrackInput
    oneTrackInput
```

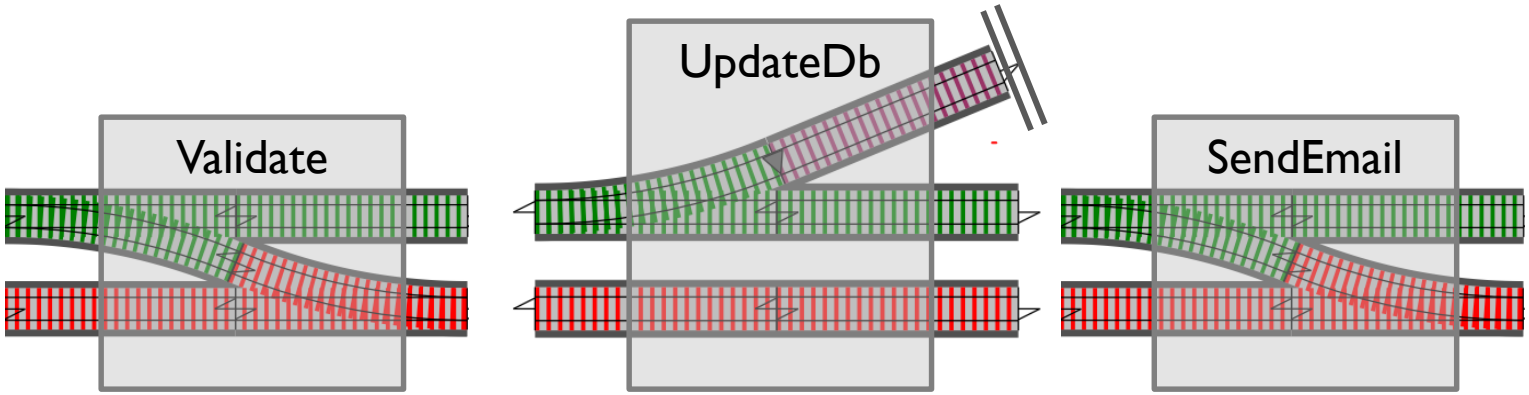
```
tee : ('a -> unit) -> 'a -> 'a
```

Dead end function

one-track input

one-track output

Converting dead-end functions



Will compose

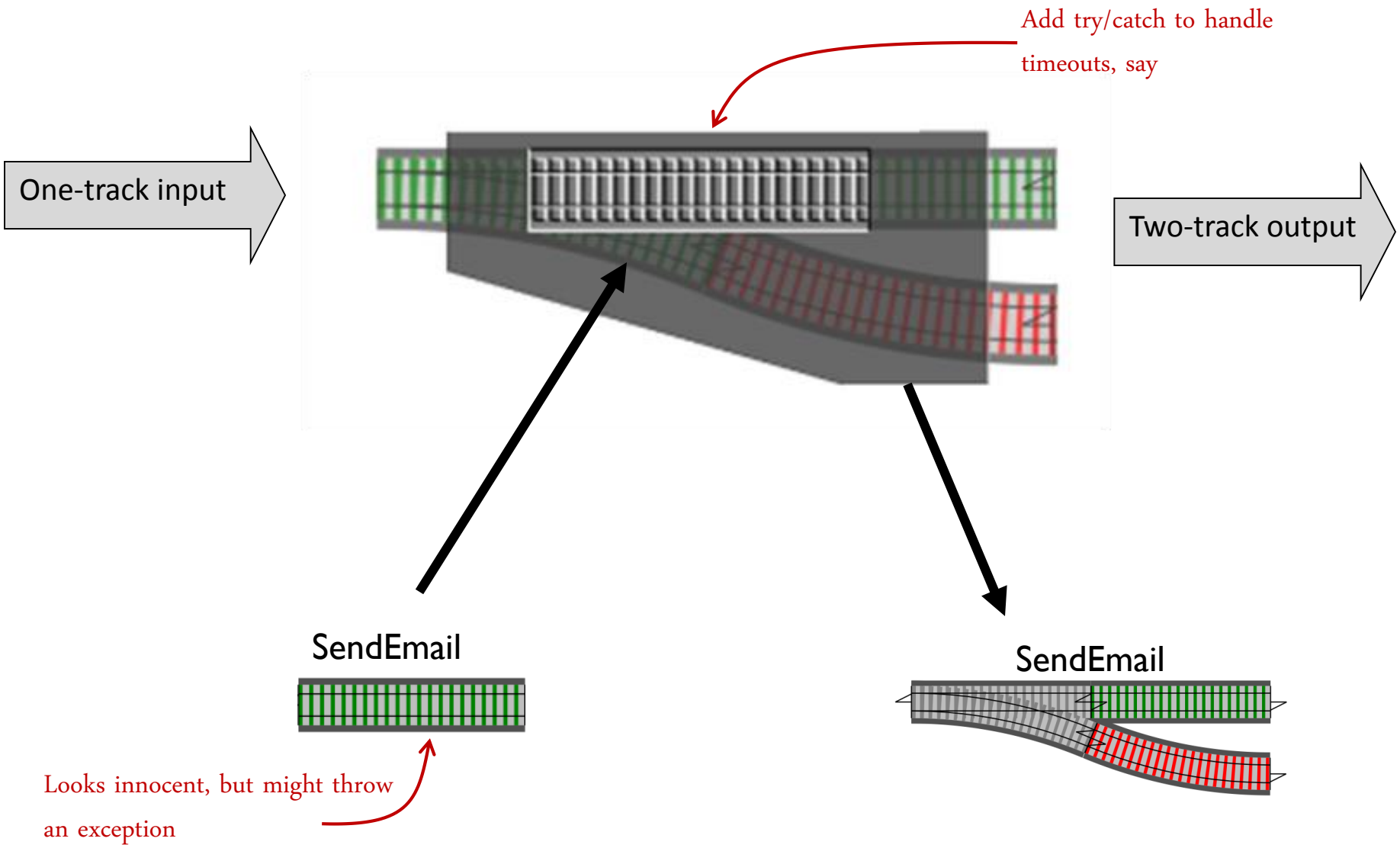
Functions that throw exceptions

Fitting other functions into this framework:

- Single track functions
- Dead-end functions
- **Functions that throw exceptions**
- Supervisory functions

Especially to wrap an I/O call

Functions that throw exceptions



Don't bother handling all possible exceptions:

E.g. FileNotFound, yes. OutOfMemory, no.

Guideline: Convert exceptions into Failures

Even Yoda recommends not to use exception handling for control flow:

"Do or do not, there is no try".

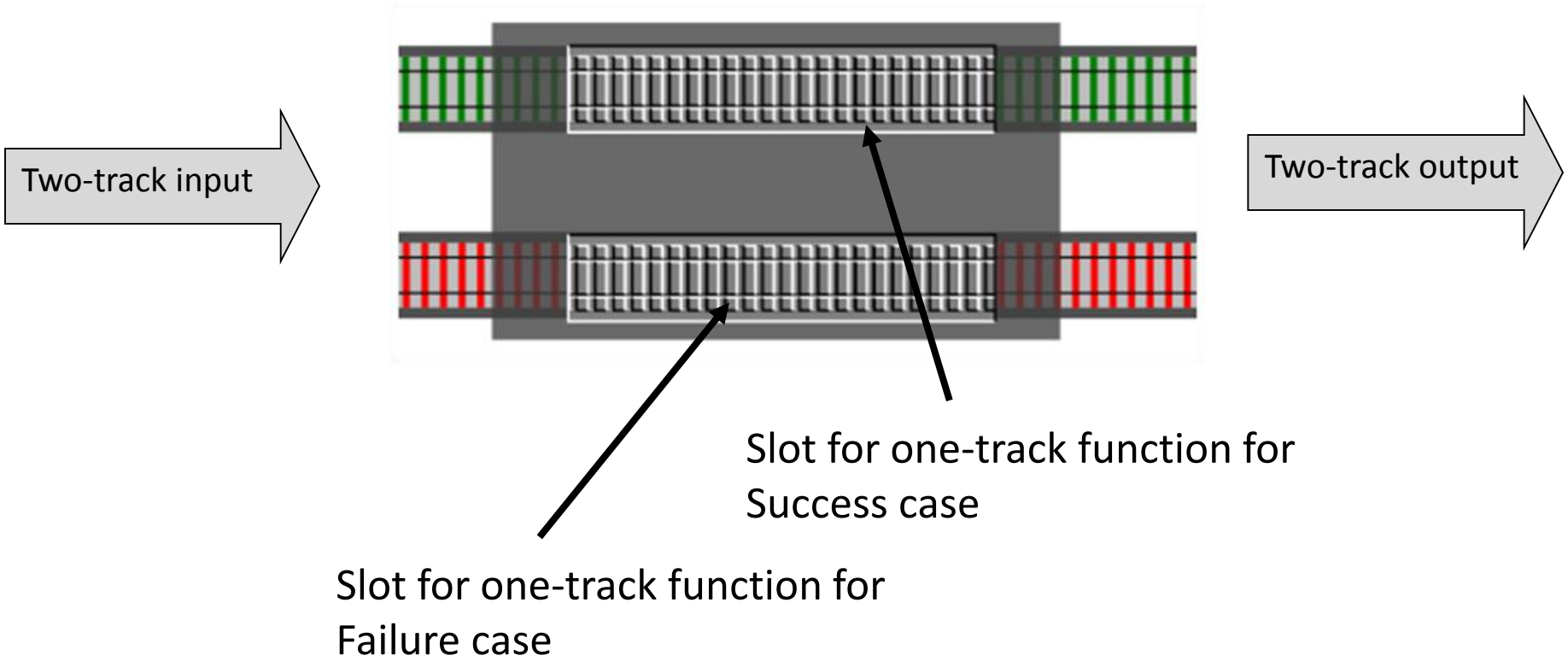
Supervisory functions

Fitting other functions into this framework:

- Single track functions
- Dead-end functions
- Functions that throw exceptions
- **Supervisory functions**

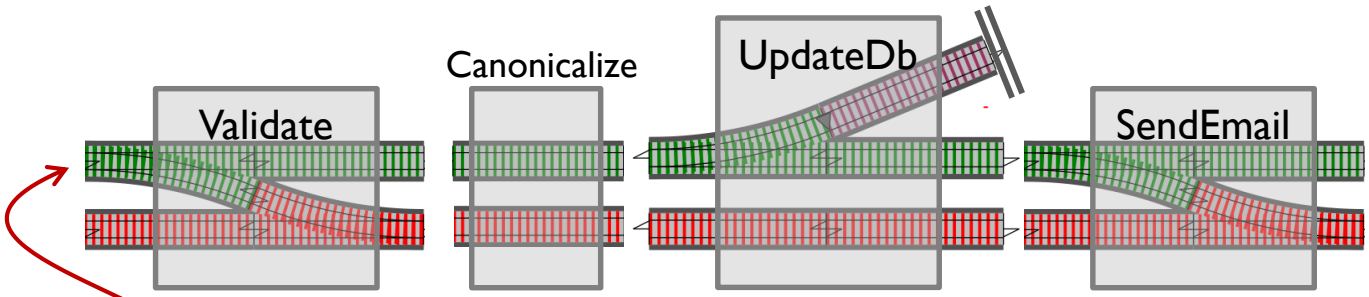
For when you need to handle *both* tracks – e.g. tracing, logging, etc.

Supervisory functions



Putting it all together

Putting it all together

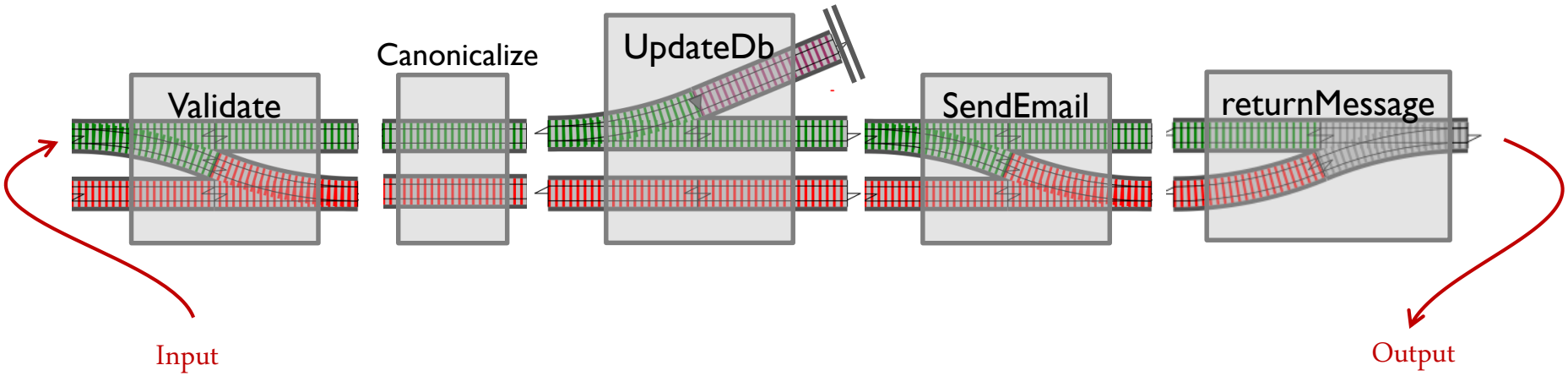


Input

Output??

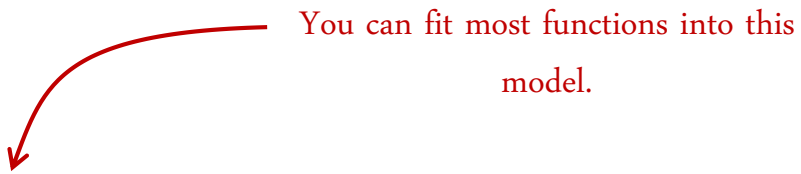
your browser doesn't understand two-track types!

Putting it all together

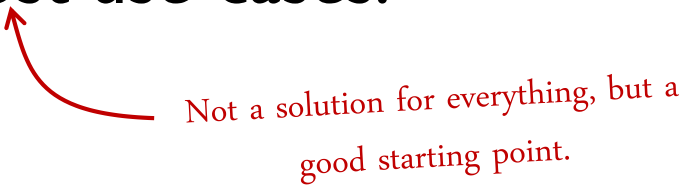



```
let returnMessage result =
  match result with
  | Success _ -> "Success"
  | Failure msg -> msg
```

Putting it all together - review



Summary: The "two-track" framework is a useful approach for most use-cases.



Putting it all together - review

The "two-track" framework is a useful approach for most use-cases.

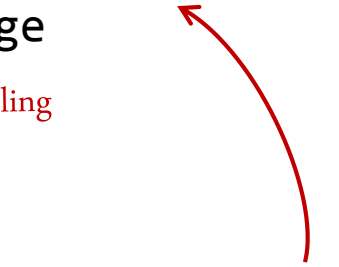
Let's look at the code -- before and after adding error handling

```
let executeUseCase =
  receiveRequest
  >> validateRequest
  >> updateDbFromRequest
  >> sendEmail
  >> returnMessage
```

Before – without error handling

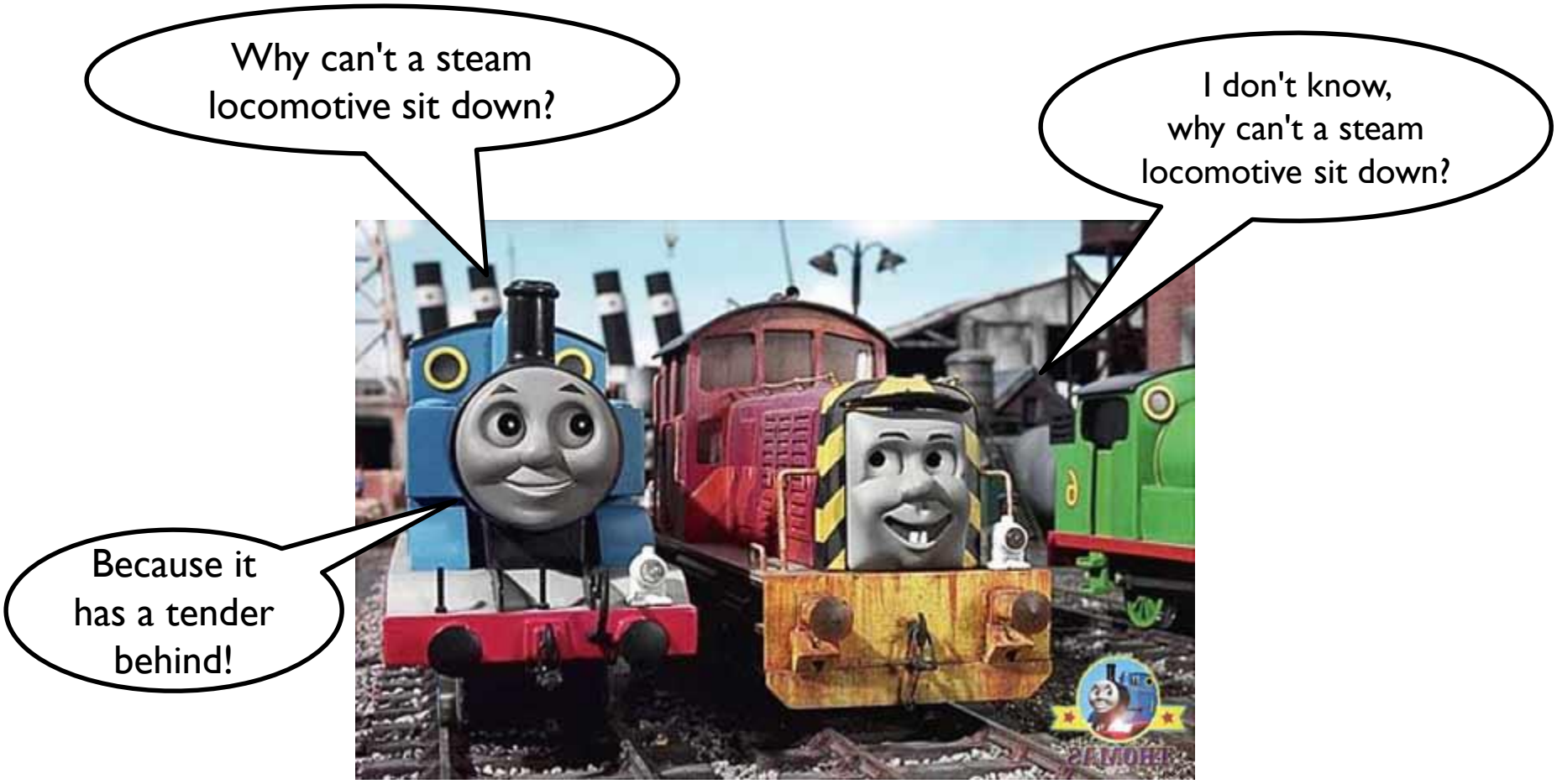
```
let executeUseCase =
  receiveRequest
  >> validateRequest
  >> updateDbFromRequest
  >> sendEmail
  >> returnMessage
```

After – with error handling



Still clean and elegant

Comic Interlude



Designing for errors

Unhappy paths are requirements too

Designing for errors

```
let validateInput input =
    if input.name = "" then
        Failure "Name must not be blank"
    else if input.email = "" then
        Failure "Email must not be blank"
    else
        Success input  // happy path

type Result<'TEntity> =
    | Success of 'TEntity
    | Failure of string
```

Designing for errors

```
let validateInput input =
    if input.name = "" then
        Failure NameMustNotBeBlank
    else if input.email = "" then
        Failure EmailMustNotBeBlank
    else
        Success input  // happy path


type Result<'TEntity> =
    | Success of 'TEntity
    | Failure of ErrorMessage

type ErrorMessage =
    | NameMustNotBeBlank
    | EmailMustNotBeBlank
```

Special type rather than string

Designing for errors

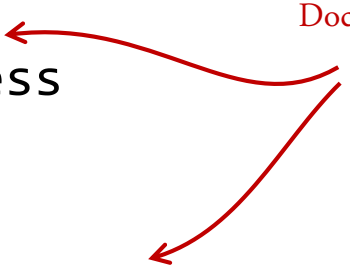
```
let validateInput input =
    if input.name = "" then
        Failure NameMustNotBeBlank
    else if input.email = "" then
        Failure EmailMustNotBeBlank
    else if (input.email doesn't match regex) then
        Failure EmailNotValid input.email
    else
        Success input  // happy path
```

Add invalid email as data

```
type ErrorMessage =
    | NameMustNotBeBlank
    | EmailMustNotBeBlank
    | EmailNotValid of EmailAddress
```

Designing for errors

```
type ErrorMessage =
   | NameMustNotBeBlank
   | EmailMustNotBeBlank
   | EmailNotValid of EmailAddress
```



Documentation of everything that can go wrong --

And it's type-safe documentation that can't go out of date!

Also triggers important DDD conversations

Designing for errors – service boundaries

Generic database errors

```
type DbErrorMessage<'PK> =
   | PrimaryKeyNotValid of 'PK
   | RecordNotFoundError of 'PK
   | DbTimeout of ConnectionString * TimeoutMs
   | DbConcurrencyError
   | DbAuthorizationError of Credentials
```

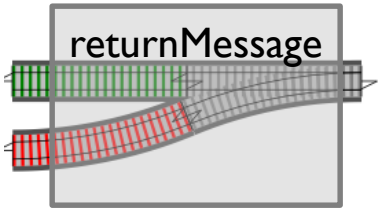
Translation function needed at a service boundary

Specific errors for this use-case

```
type MyUseCaseError =
   | NameMustNotBeBlank
   | EmailMustNotBeBlank
   | EmailNotValid of EmailAddress
   // database errors
   | UserIdNotValid of UserId
   | DbUserNotFoundError of UserId
   | DbTimeout of ConnectionString
   | DbConcurrencyError
   | DbAuthorizationError of Credentials
   // SMTP errors
   | SmtpTimeout of SmtpConnection
   | SmtpBadRecipient of EmailAddress
```

```
let dbResultToMyResult dbError =
  match dbError with
  | DbErrorMessage.PrimaryKeyNotValid id ->
    MyUseCaseError.UserIdNotValid id
  | DbErrorMessage.RecordNotFoundError id ->
    MyUseCaseError.DbUserNotFoundError id
  | _  -> // etc
```

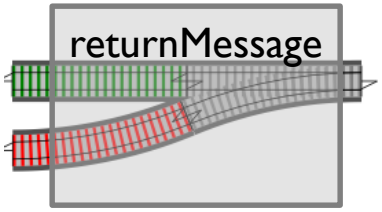
Designing for errors – converting to strings




```
let returnMessage result =
  match result with
  | Success _ -> "Success"
  | Failure msg -> msg
```

No longer works – each case must now be explicitly converted to a string

Designing for errors – converting to strings




```
let returnMessage result =
  match result with
  | Success _ -> "Success"
  | Failure err ->
      match err with
      | NameMustNotBeBlank -> "Name must not be blank"
      | EmailMustNotBeBlank -> "Email must not be blank"
      | EmailNotValid (EmailAddress email) ->
          sprintf "Email %s is not valid" email

      // database errors
      | UserIdNotValid (UserId id) ->
          sprintf "User id %i is not a valid user id" id
      | DbUserNotFoundError (UserId id) ->
          sprintf "User id %i was not found in the database" id
      | DbTimeout (_,TimeoutMs ms) ->
          sprintf "Could not connect to database within %i ms" ms
      | DbConcurrencyError ->
          sprintf "Another user has modified the record. Please resubmit"
      | DbAuthorizationError _ ->
          sprintf "You do not have permission to access the database"

      // SMTP errors
      | SmtpTimeout (_,TimeoutMs ms) ->
          sprintf "Could not connect to SMTP server within %i ms" ms
      | SmtpBadRecipient (EmailAddress email) ->
          sprintf "The email %s is not a valid recipient" email
```

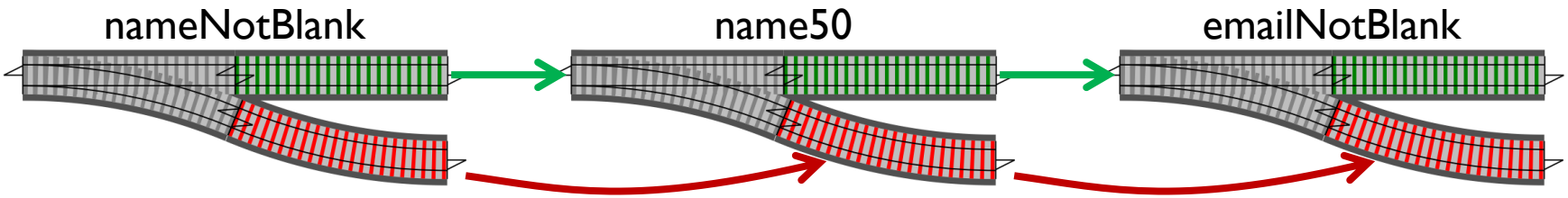
Each case must be converted to a string – but this is only needed once, and only at the last step.

Different conversions can be used depending on the target. E.g. user messages vs. logging.

All strings are in one place, so translations are easier. (or use resource file)

Parallel tracks

Parallel validation

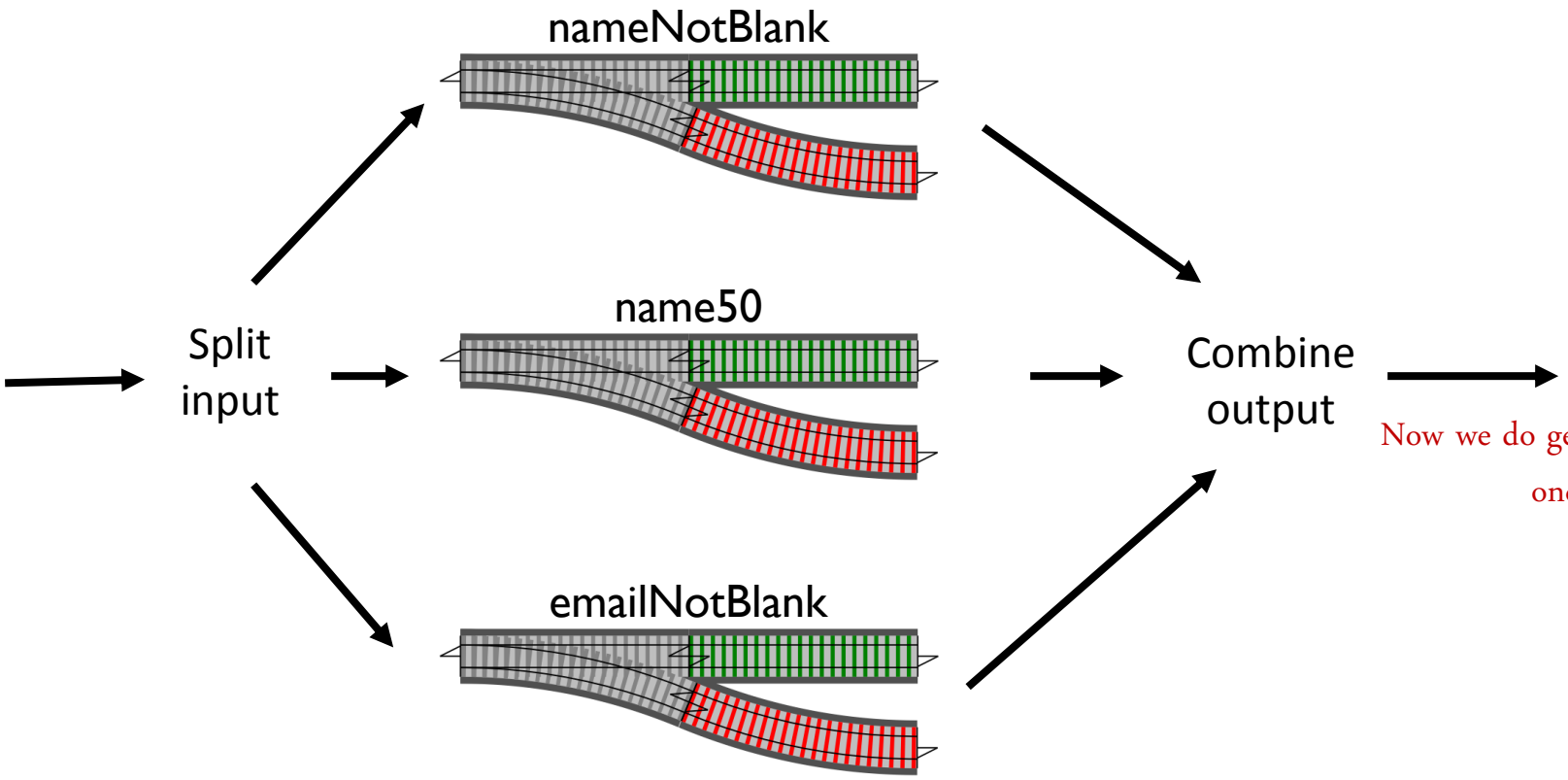


Problem: Validation done in series.

So only one error at a time is returned

It would be nice to return all validation errors at once.

Parallel validation

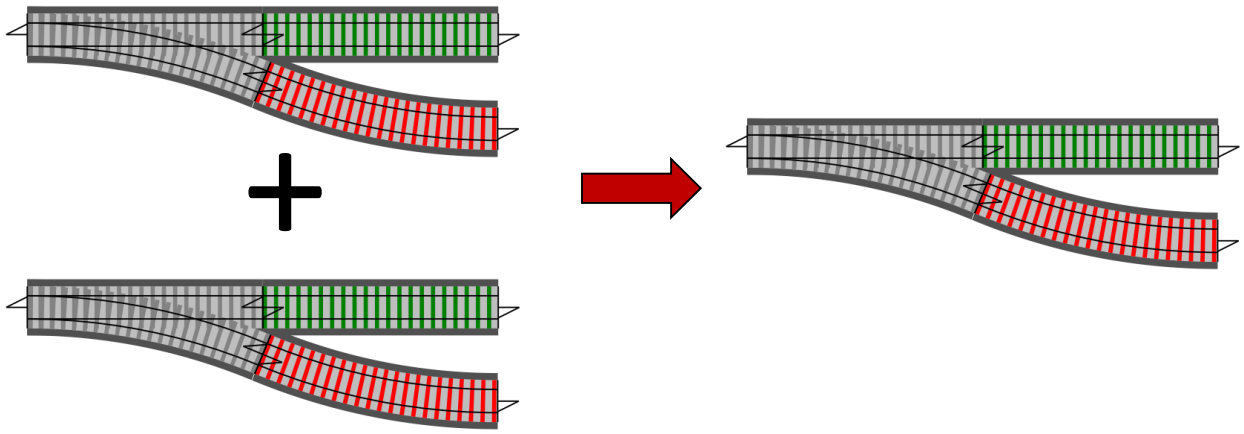


Now we do get all errors at once!

... But how to combine?

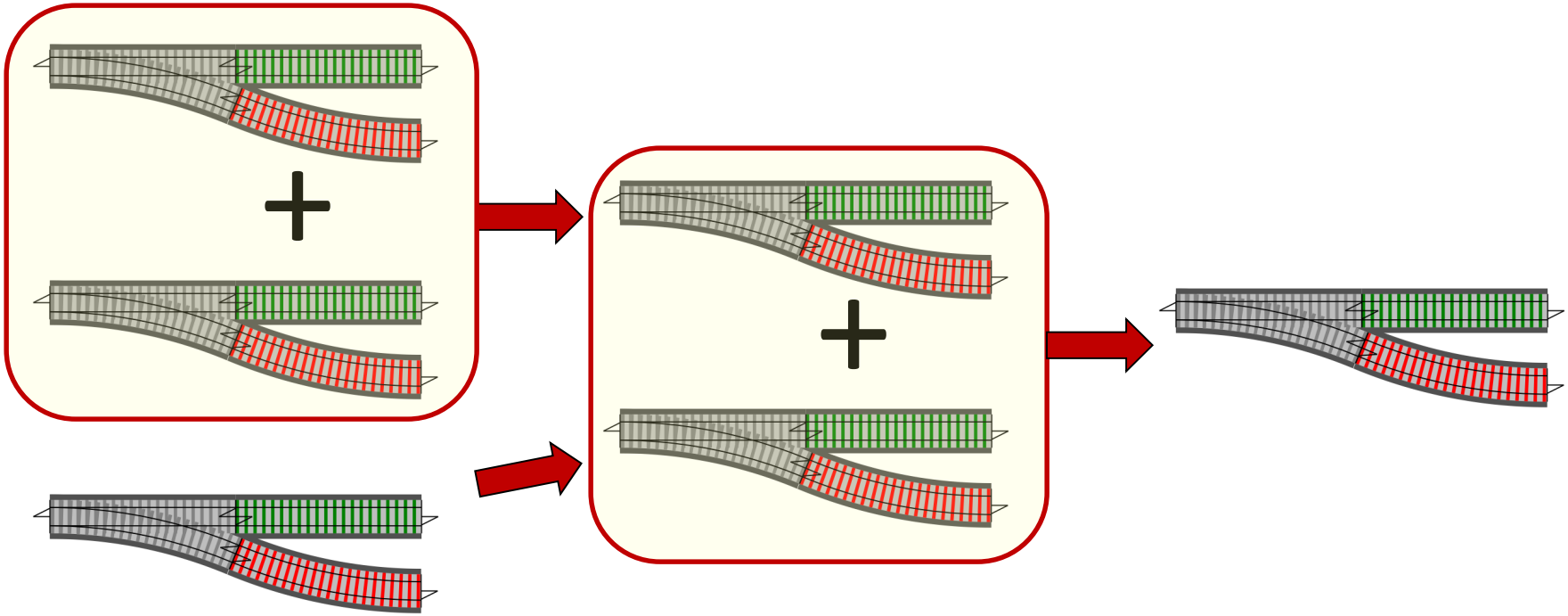
Combining switches

Trick: if we create an operation that combines pairs into a new switch, we can repeat to combine as many switches as we like.



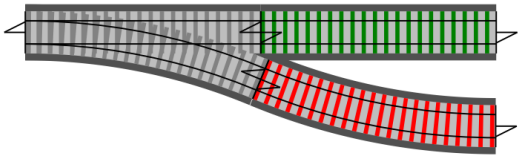
Combining switches

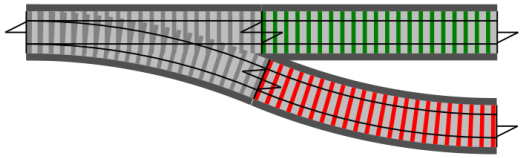
Trick: if we create an operation that combines pairs into a new switch, we can repeat to combine as many switches as we like.



-> For more, see "monoids without tears"

Combining switches

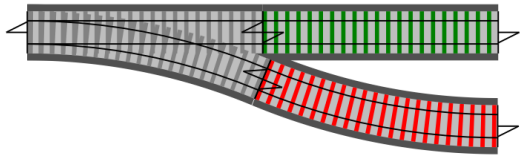




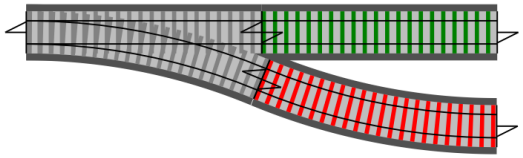
+	Success (S2)	Failure (F2)
Success (S1)	S1 or S2	F2
Failure (F1)	F1	[F1; F2]

A failure in either one is a overall failure.

Combining switches

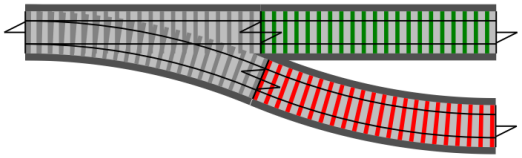


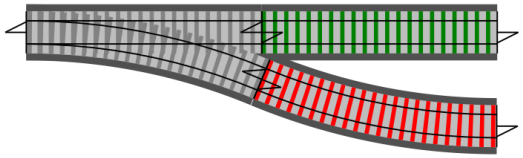
Either input is OK, they are both the same value



+	Success (S2)	Failure (F2)
Success (S1)	S1 or S2	F2
Failure (F1)	F1	[F1; F2]

Combining switches



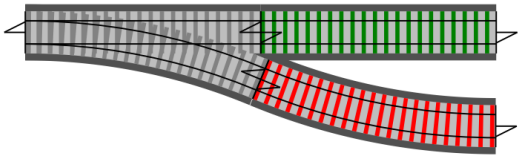


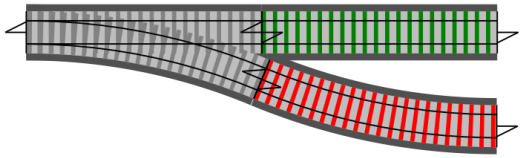
+	Success (S2)	Failure (F2)
Success (S1)	S1 or S2	F2
Failure (F1)	F1	[F1; F2]

We need to keep both, so store in a list.

```
type Result<'TEntity> =
    | Success of 'TEntity
    | Failure of ErrorMessage list
```

Combining switches



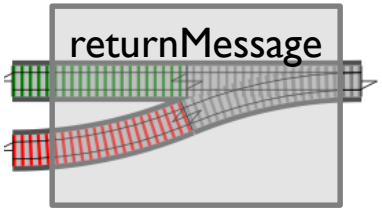


+	Success (S2)	Failure (F2)
Success (S1)	S1 or S2	[F2]
Failure (F1)	[F1]	[F1; F2]

But now these need to be lists too.

```
type Result<'TEntity> =
    | Success of 'TEntity
    | Failure of ErrorMessage list
```

Handling lists of errors




```
let errToString err =
  match err with
  | NameMustNotBeBlank -> "Name must not be blank"
  | EmailMustNotBeBlank -> "Email must not be blank"
  // etc

let returnMessage result =
  match result with
  | Success _ -> "Success"
  | Failure errs ->
      errs
      |> List.map errToString
      |> List.reduce (fun s1 s2 -> s1 + ";" + s2)
```

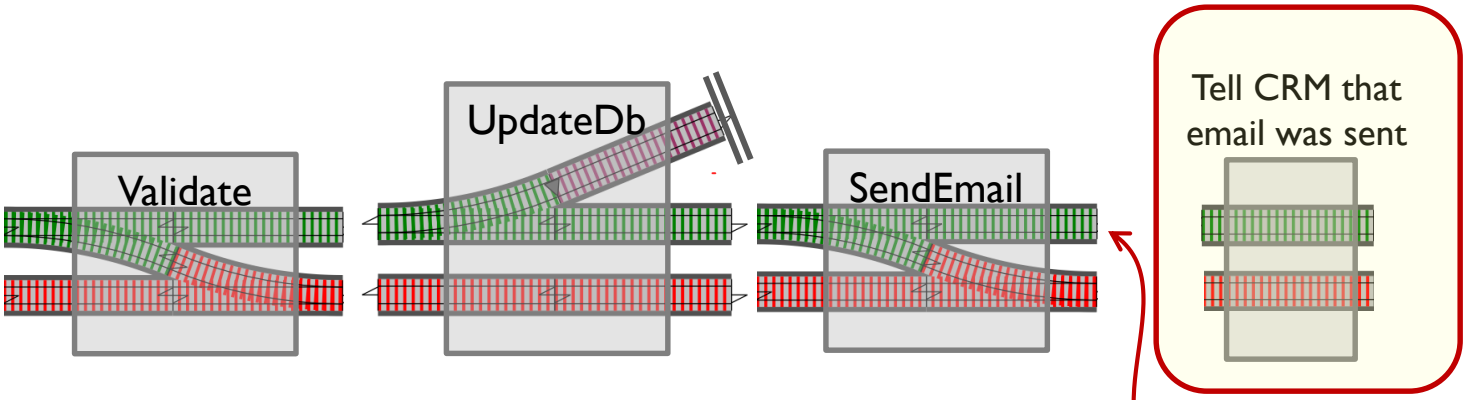
Convert all messages to strings

Collapse a list of strings into a single string

Domain events

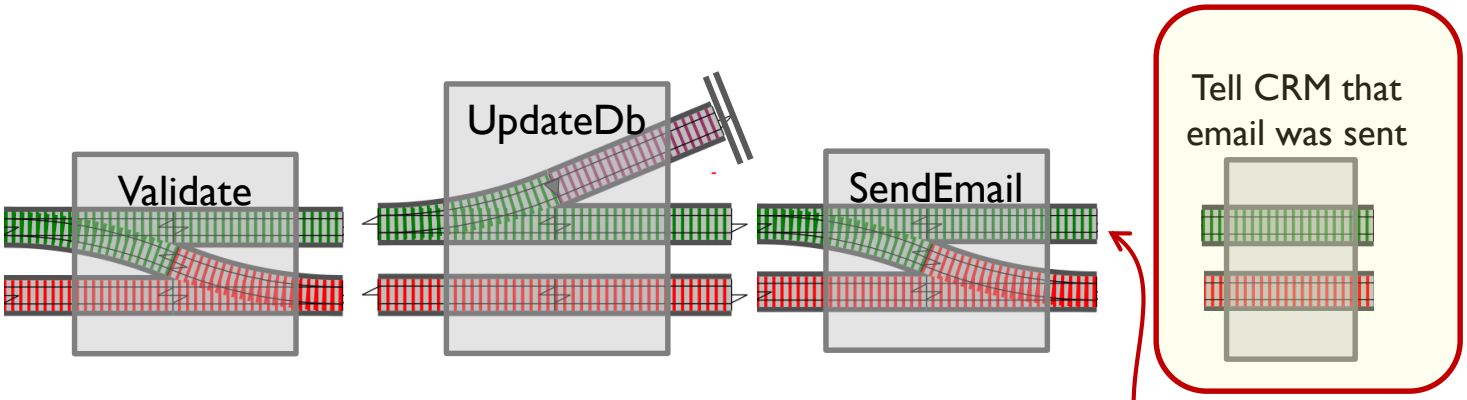
Communicating information to downstream functions

Events are not errors



Communicate new information to downstream processes without being a failure

Events are not errors

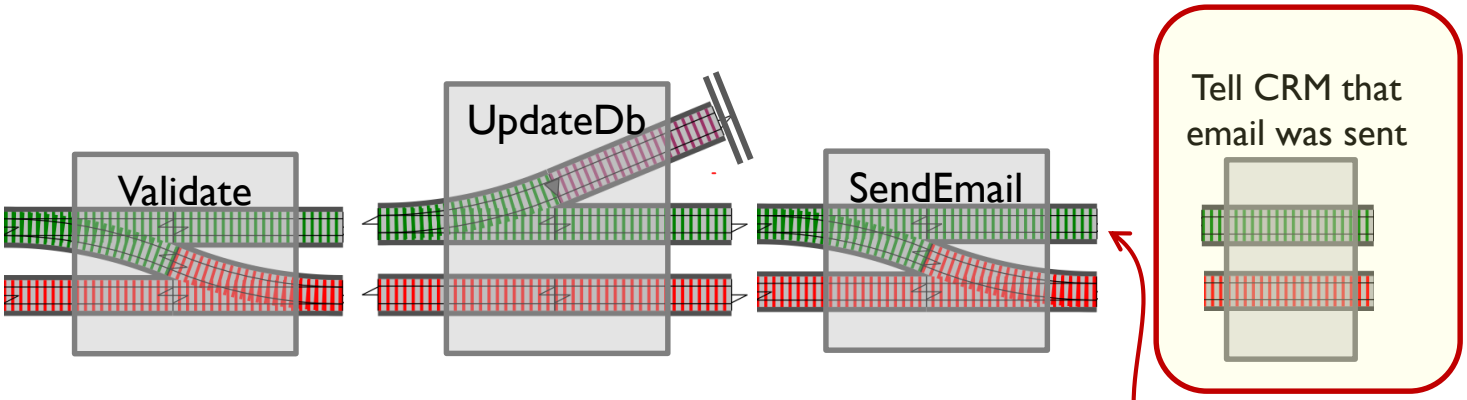



```
type MyUseCaseMessage =
  | NameMustNotBeBlank
  | EmailMustNotBeBlank
  | EmailNotValid of EmailAddress
  // database errors
  | UserIdNotValid of UserId
  // SMTP errors
  | SmtpTimeout of SmtpConnection
  // Domain events
  | UserSaved of AuditInfo
  | EmailSent of EmailAddress * MsgId
```

Communicate new information to downstream processes without being a failure

Add domain events to the choices

Events are not errors



Communicate new information to downstream processes without being a failure

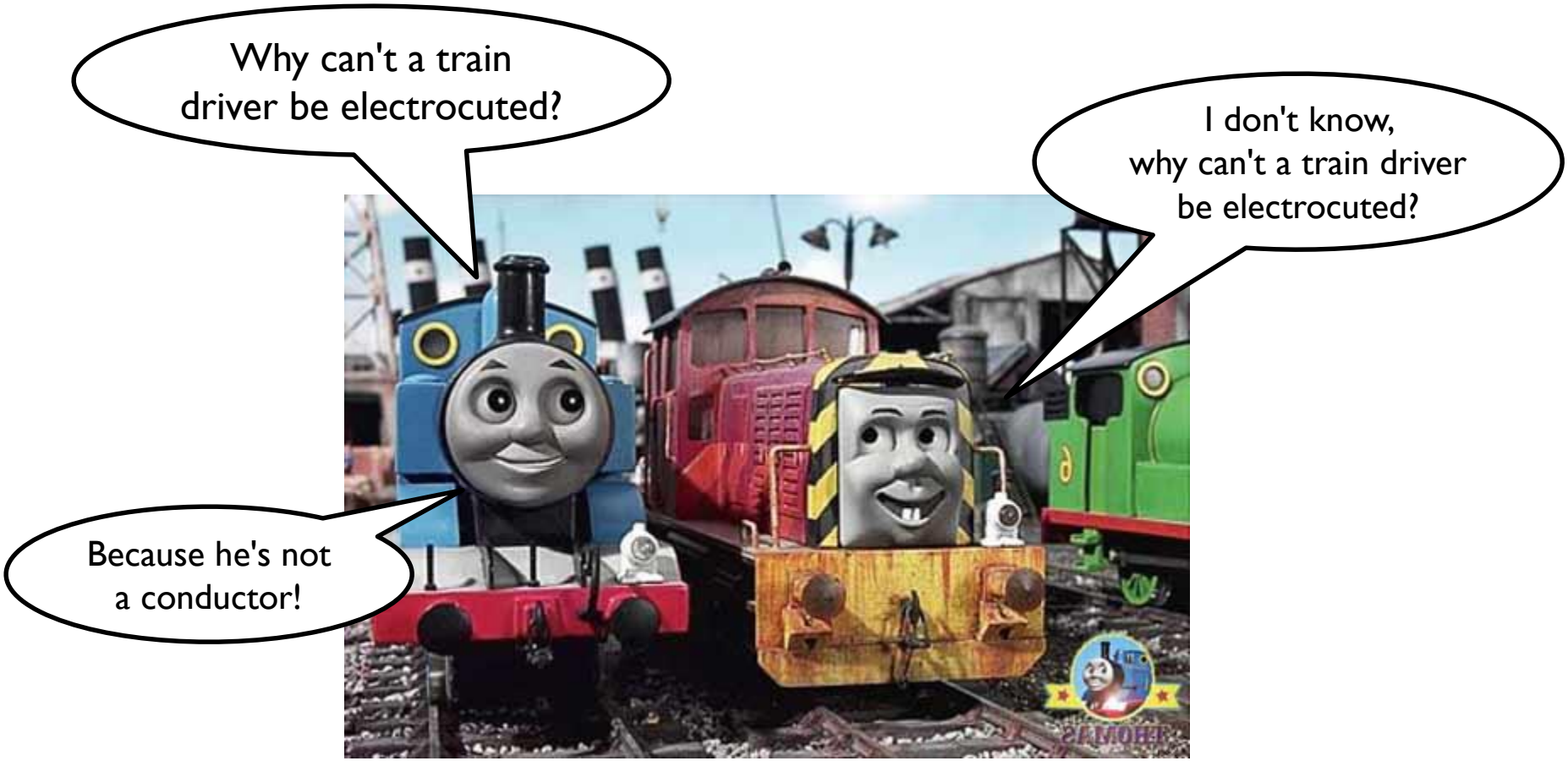
```
type MyUseCaseMessage =
  | NameMustNotBeBlank
  | EmailMustNotBeBlank
  | EmailNotValid of EmailAddress
  // database errors
  | UserIdNotValid of UserId
  // SMTP errors
  | SmtpTimeout of SmtpConnection
  // Domain events
  | UserSaved of AuditInfo
  | EmailSent of EmailAddress * MsgId
```

Add domain events to the choices

Store a list of events (as well as the entity) on the Success path

```
type Result<'TEntity> =
    | Success of 'TEntity * Message list
    | Failure of Message list
```

Comic Interlude



Summary

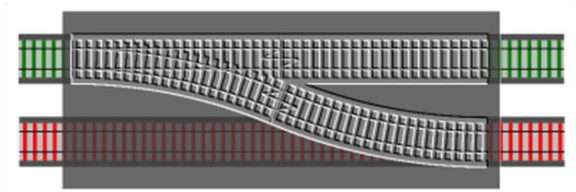
A recipe for handling errors in a functional way

Recipe for handling errors in a functional way

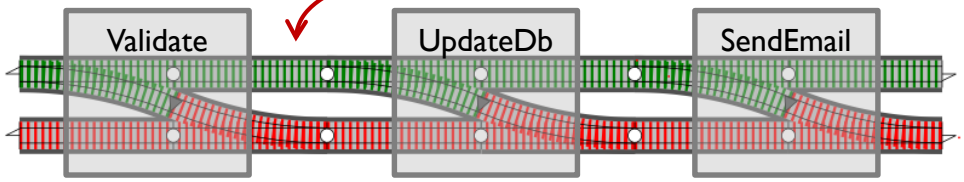
Step 1: Create a Result type

```
type Result<'TEntity> =
    | Success of 'TEntity * Message list
    | Failure of Message list
```

Step 2: Use "bind" to convert switches to two-track functions



Step 3: Use composition to glue the two-track functions together



Step 4: Make error cases first class citizens

```
type Message =
  | NameMustNotBeBlank
  | EmailMustNotBeBlank
  | EmailNotValid of EmailAddress
```

Some topics not covered...

... but could be handled
in an obvious way.

Topics not covered

- Async on success path (instead of sync)
- Compensating transactions
  (instead of two phase commit)
- Logging (tracing, app events, etc.)


I DON'T ALWAYS HAVE ERRORS
BUT WHEN I DO, I USE MONADS
memegenerator.net

Railway Oriented Programming

@ScottWlaschin

fsharpforfunandprofit.com /rop

FPbridge.co.uk

Let me know if you need help with F#

Slides will be available here & code too.