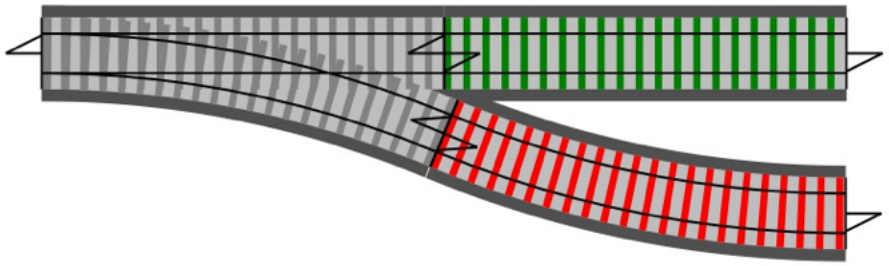
Railway Oriented Programming

A functional approach to error handling



What do railways have to do with programming?

Overview

Topics covered:

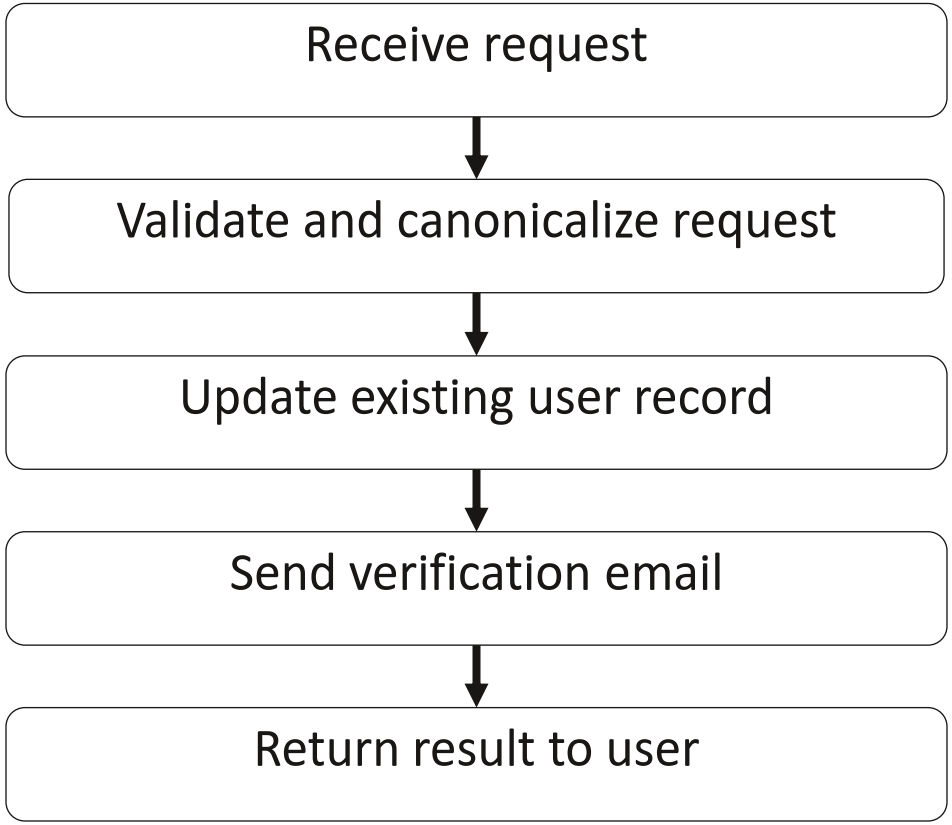
- Happy path programming
- Straying from the happy path
- Introducing "Railway Oriented Programming"
- Using the model in practice
- Extending and improving the design

Happy path programming

Implementing a simple use case

A simple use case

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



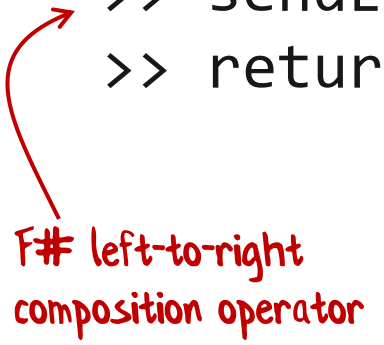

```
type Request = {
  userId: int;
  name: string;
  email: string }
```

Imperative code

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

Functional flow

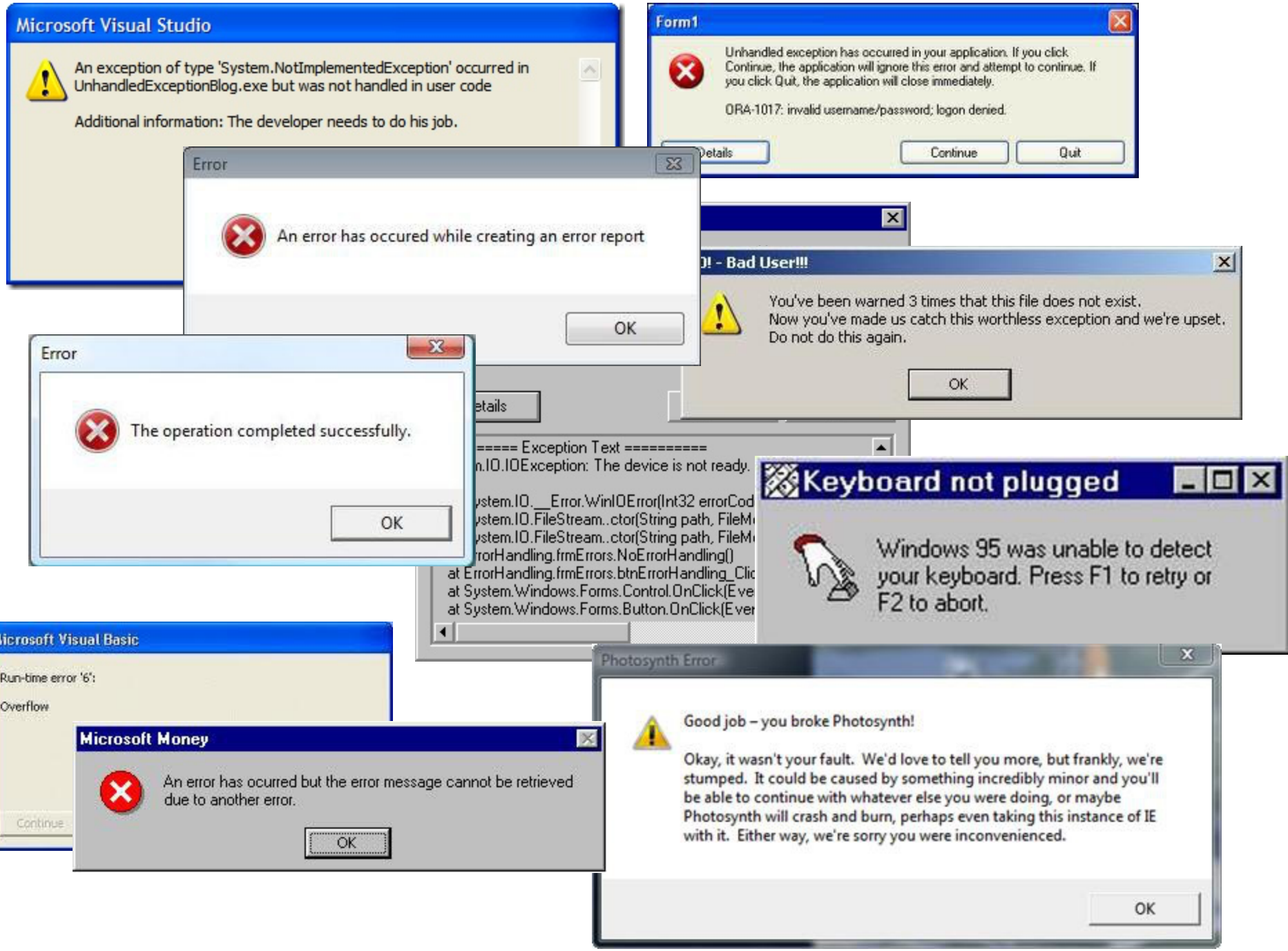
```
let executeUseCase =
  receiveRequest
  >> validateRequest
  >> canonicalizeEmail
  >> updateDbFromRequest
  >> sendEmail
  >> returnMessage
```



Straying from the happy path...

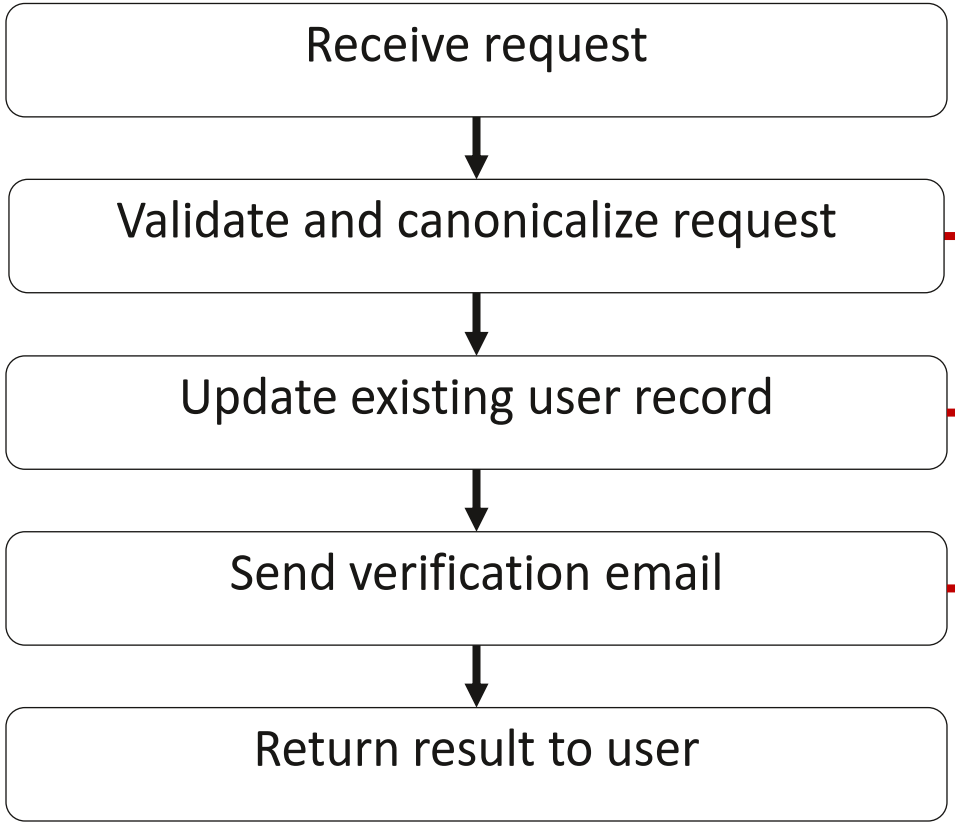
What do you do when
something goes wrong?

“A program is a spell cast over a
computer, turning input
into error messages”


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
rorHandling.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 }
```

Name is blank
Email not valid

User not found
Db error

Authorization error
Timeout

Imperative code with error handling

```
string UpdateCustomerWithErrorHandling()
{
  var request = receiveRequest();
  validateRequest(request);
  canonicalizeEmail(request);
  db.updateDbFromRequest(request);
  smtpServer.sendEmail(request.Email)

  return "OK";
}
```

Imperative code with error handling

```
string UpdateCustomerWithErrorHandling()
{
  var request = receiveRequest();
  var isValidated = validateRequest(request);
  if (!isValidated) {
     return "Request is not valid"
  }
  canonicalizeEmail(request);
  db.updateDbFromRequest(request);
  smtpServer.sendEmail(request.Email)

  return "OK";
}
```

Imperative code with error handling

```
string UpdateCustomerWithErrorHandling()
{
  var request = receiveRequest();
  var isValidated = validateRequest(request);
  if (!isValidated) {
     return "Request is not valid"
  }
  canonicalizeEmail(request);
  var result = db.updateDbFromRequest(request);
  if (!result) {
    return "Customer record not found"
  }

  smtpServer.sendEmail(request.Email)

  return "OK";
}
```

Imperative code with error handling

```
string UpdateCustomerWithErrorHandling()
{
  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"
  }

  smtpServer.sendEmail(request.Email)

  return "OK";
}
```

Imperative code with error handling

```
string UpdateCustomerWithErrorHandling()
{
  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";
}
```

Imperative code with error handling

```
string UpdateCustomerWithErrorHandling()
{
  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.

Functional flow with error handling


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

Q: What is the functional equivalent of this code?

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

Functional flow with error handling

Does this look familiar?

Don't believe me? Stay tuned!

After

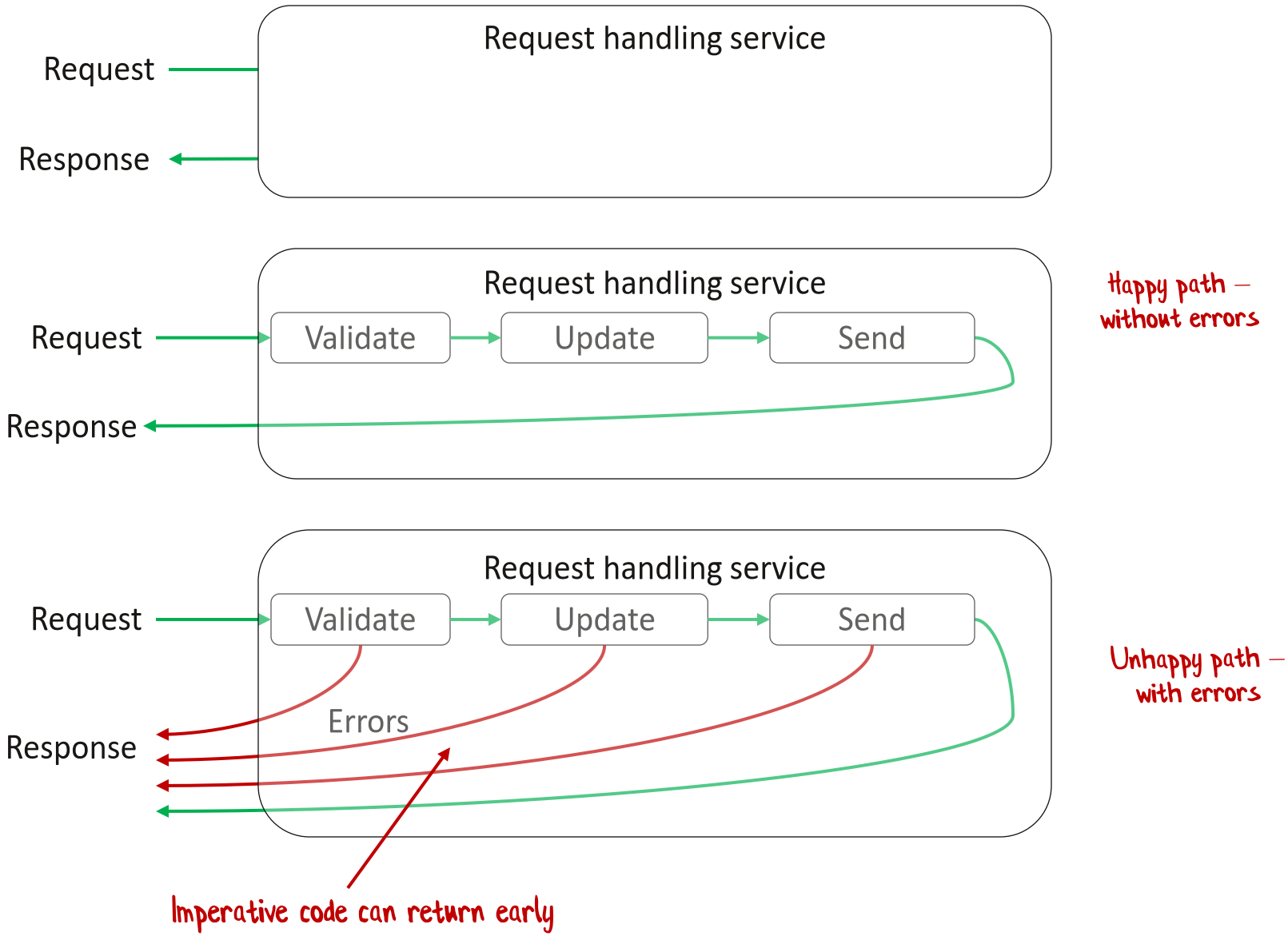
```
let updateCustomerWithErrorHandling =
  receiveRequest
  >> validateRequest
  >> canonicalizeEmail
  >> updateDbFromRequest
  >> sendEmail
  >> returnMessage
```

Q: What is the functional equivalent of this code?

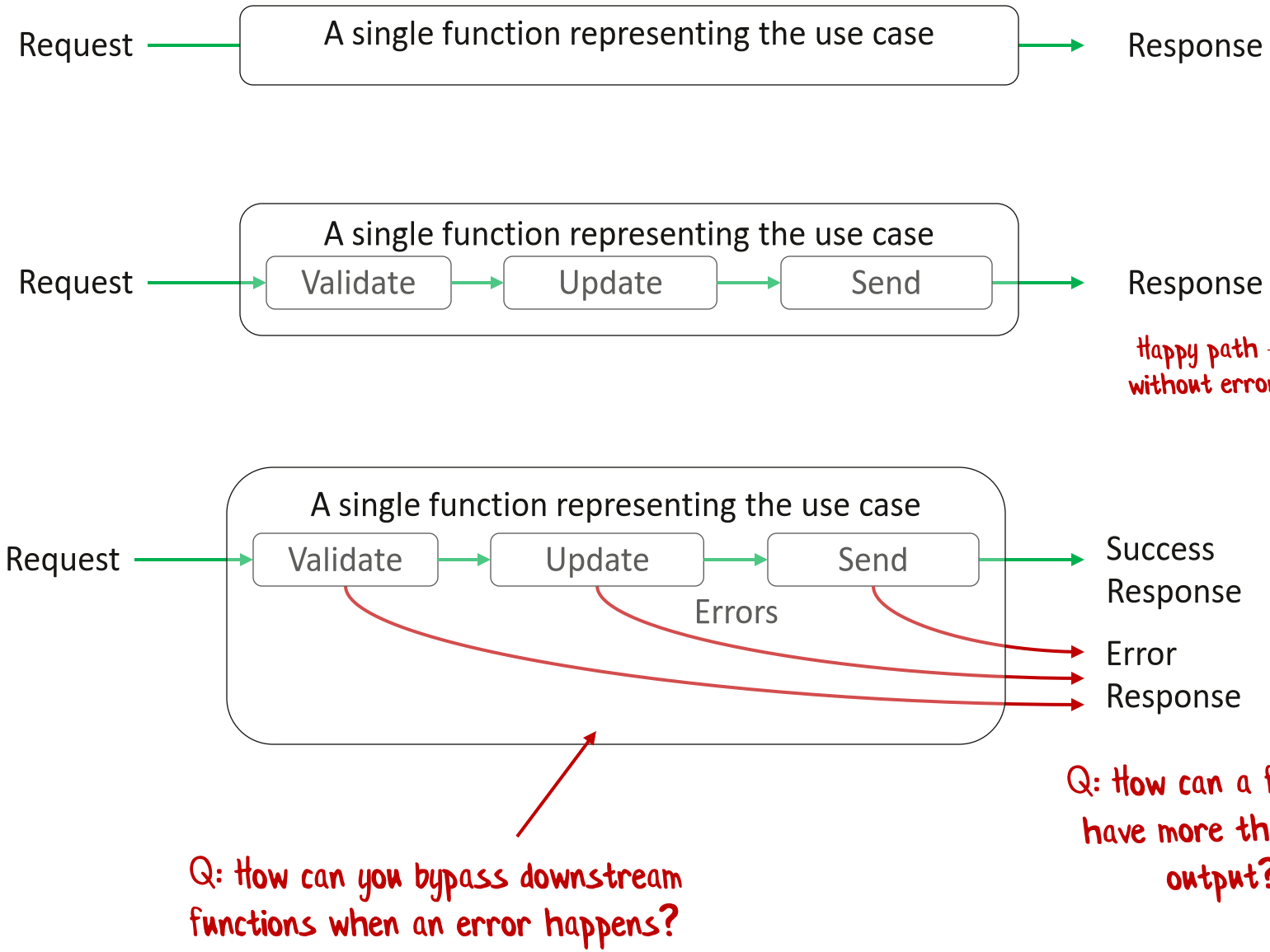
... and can we preserve the elegance of the original functional version?

Designing the unhappy path

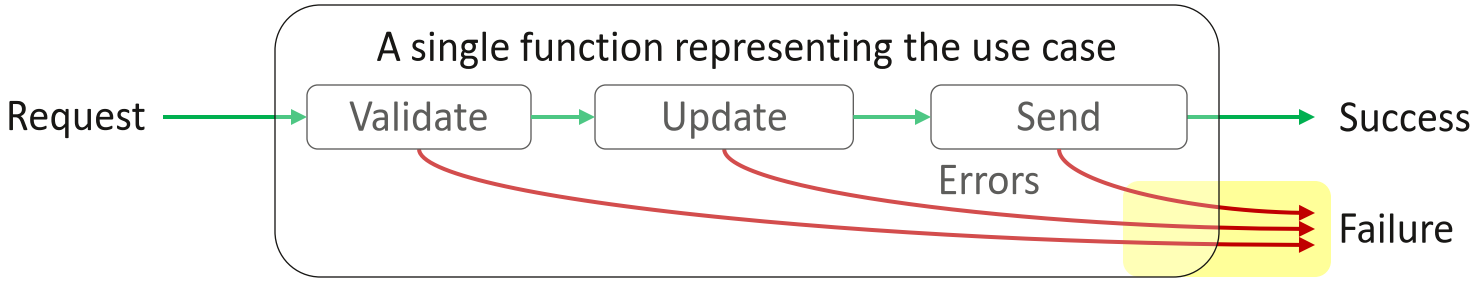
Request/response (non-functional) design



Data flow (functional) design



Functional design



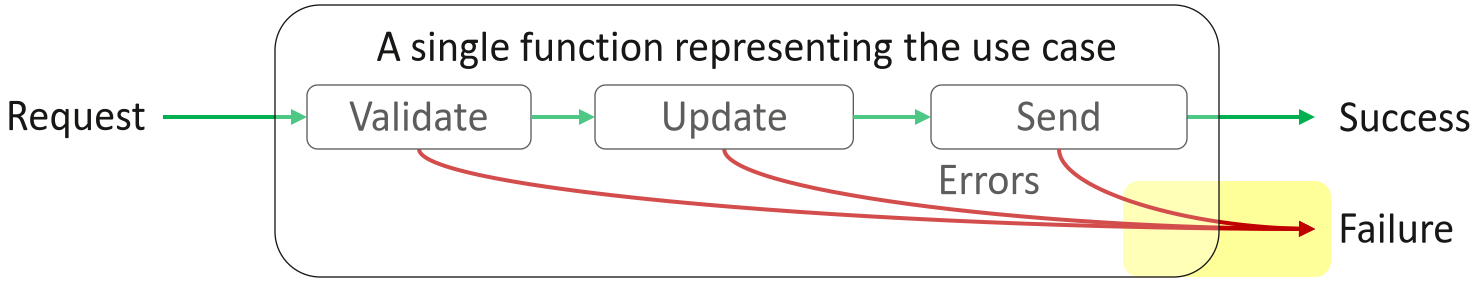
How can a function have more than one output?

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

I love sum types!

But maybe too specific for this case?

Functional design

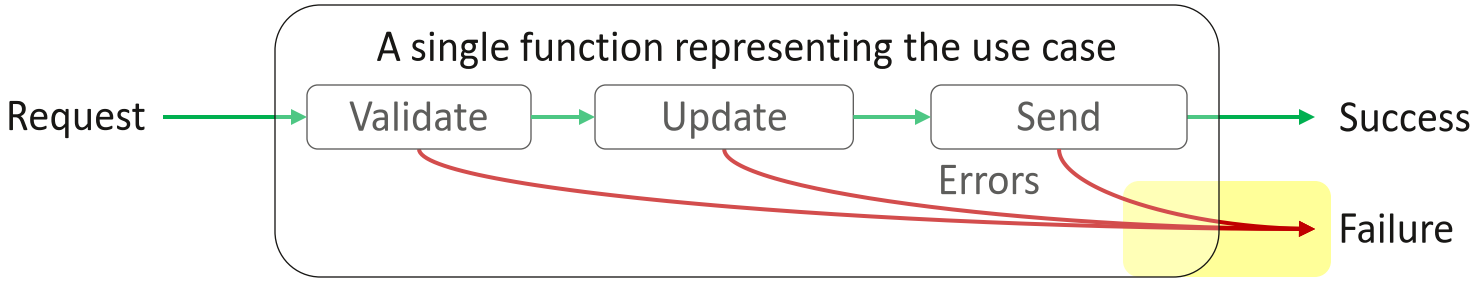


How can a function have more than one output?

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

Much more generic – but no data!

Functional design

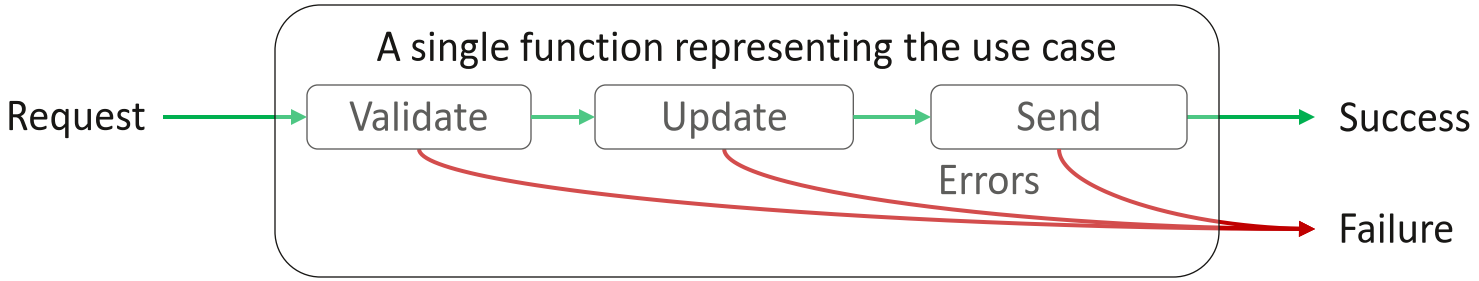


How can a function have more than one output?

```
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.

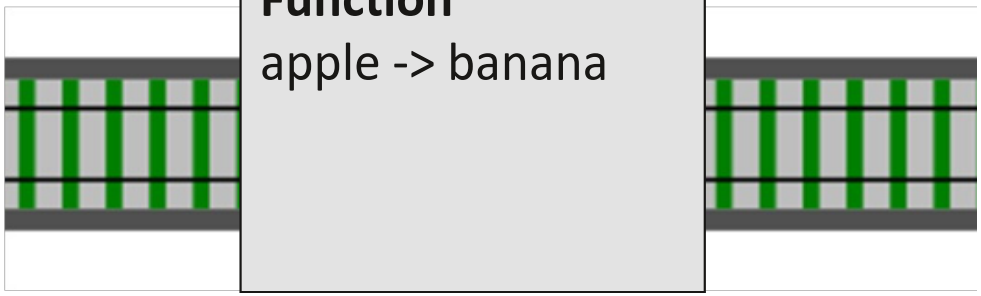
But we haven't answered the question:
How can you bypass downstream functions
when an error happens?

How do I work with errors in a functional way?

Railway oriented programming

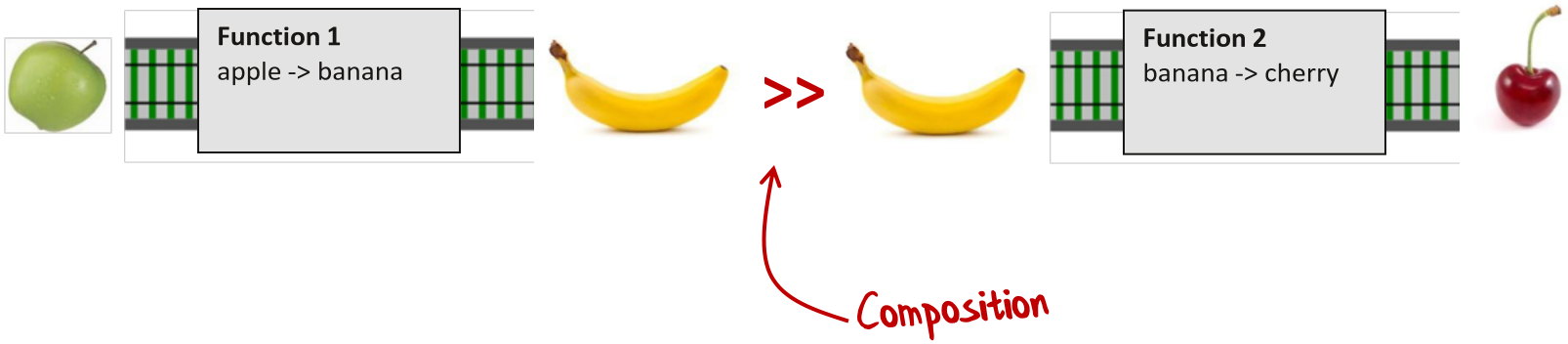
This has absolutely nothing to do with monads.

A railway track analogy

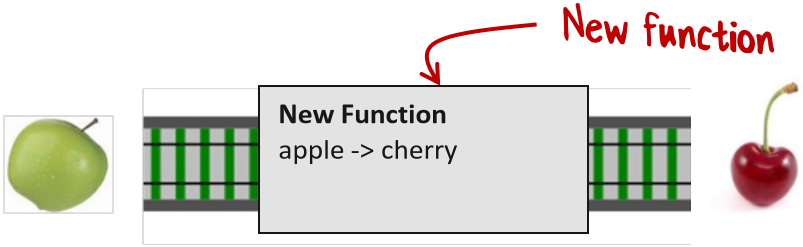


A railway track analogy

A railway track analogy



A railway track analogy



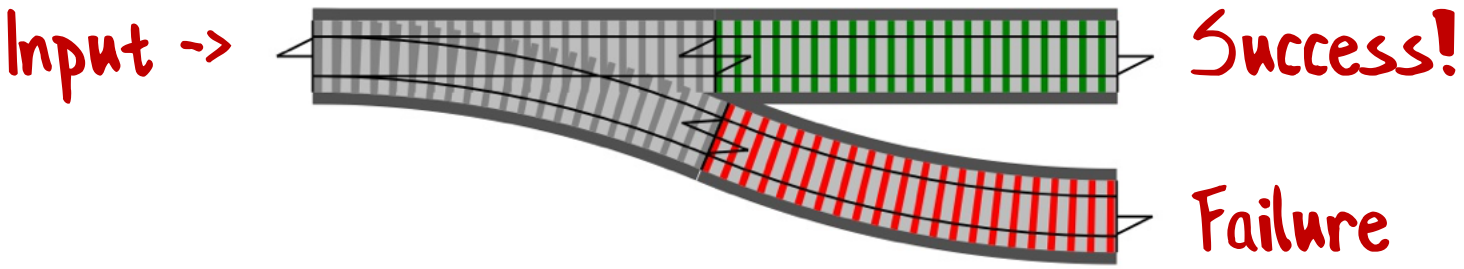
Can't tell it was built from smaller functions!

An error generating function

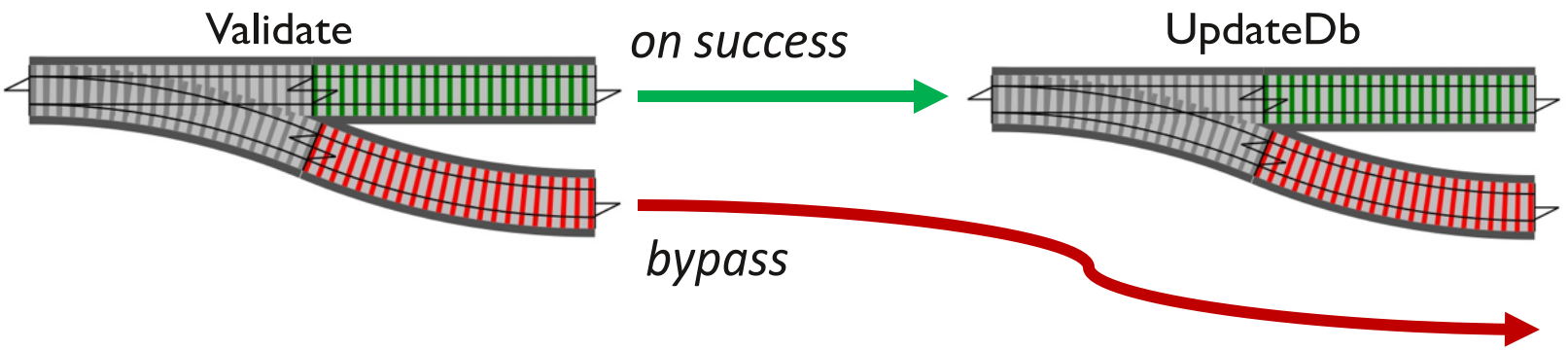
How do we model this as railway track?

```
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

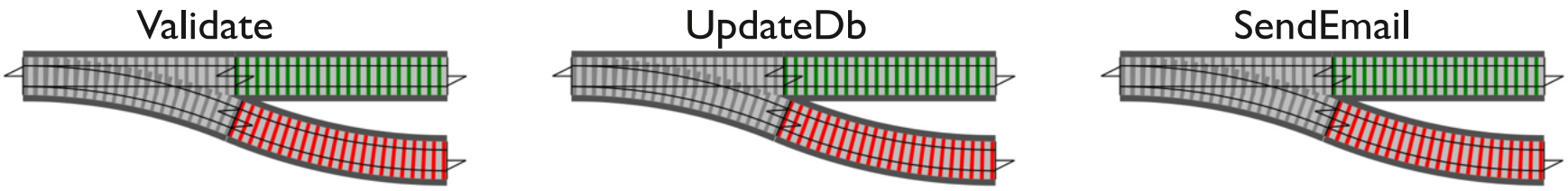


Connecting switches

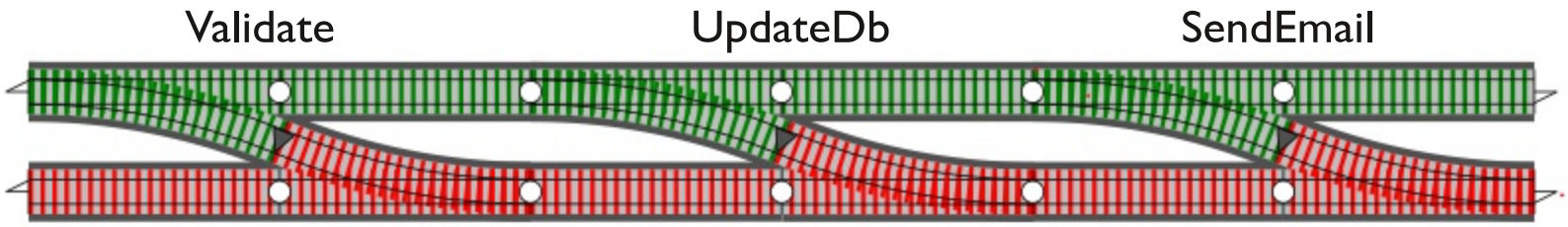


Connecting switches

Connecting switches



Connecting switches



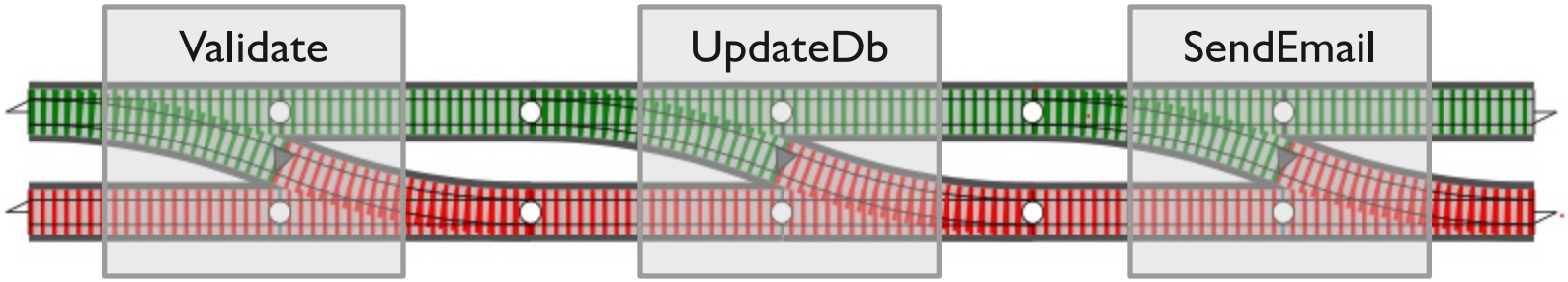
This is the "two track" model –
the basis for the "Railway Oriented Programming"
approach to error handling.

The two-track model in practice

Composing switches

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

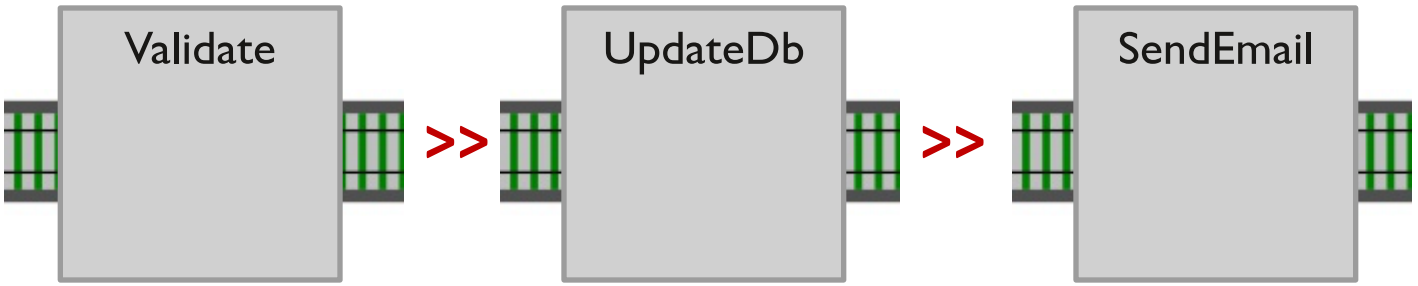
Composing switches



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

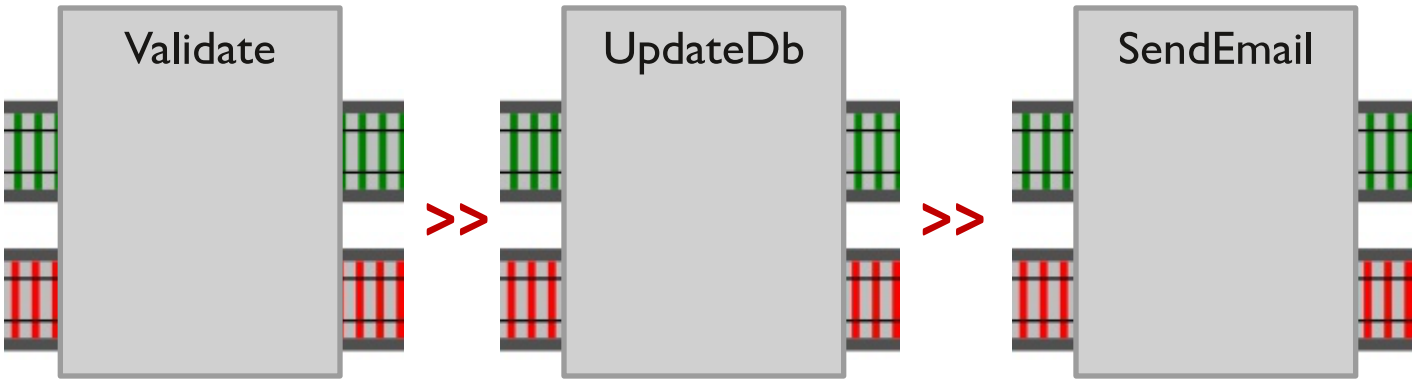
Inside each box there is a switch function.

Composing switches



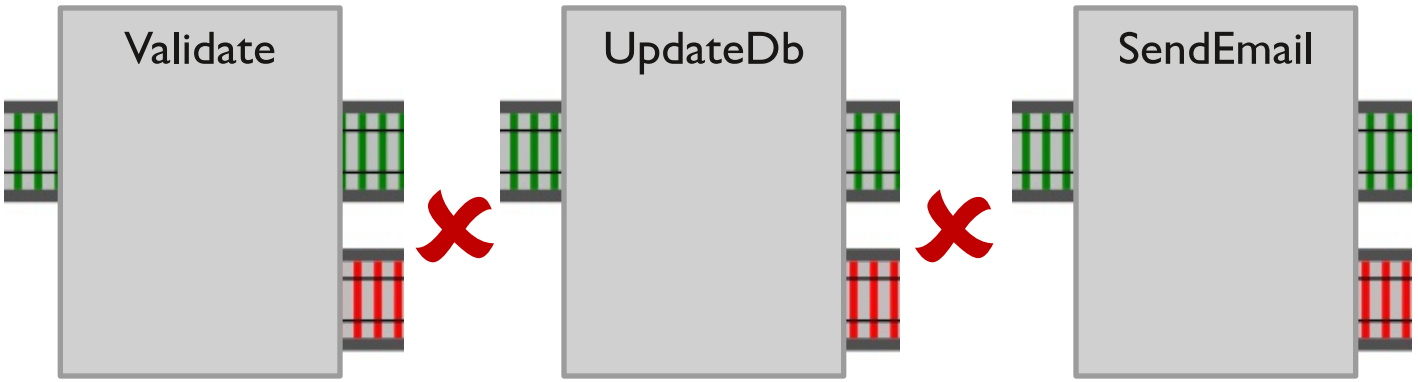
Composing one-track functions is fine...

Composing switches



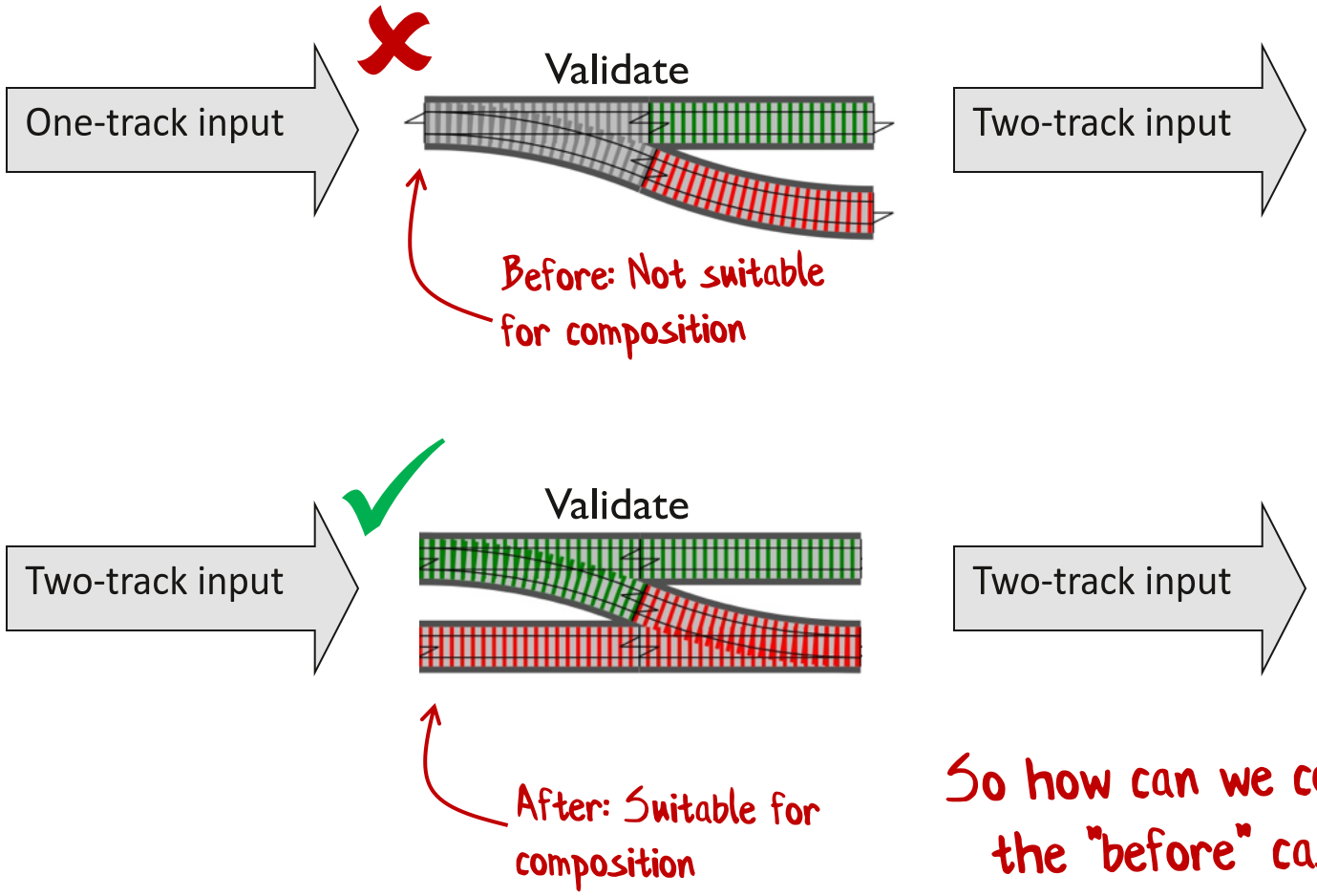
... and composing two-track functions is fine...

Composing switches

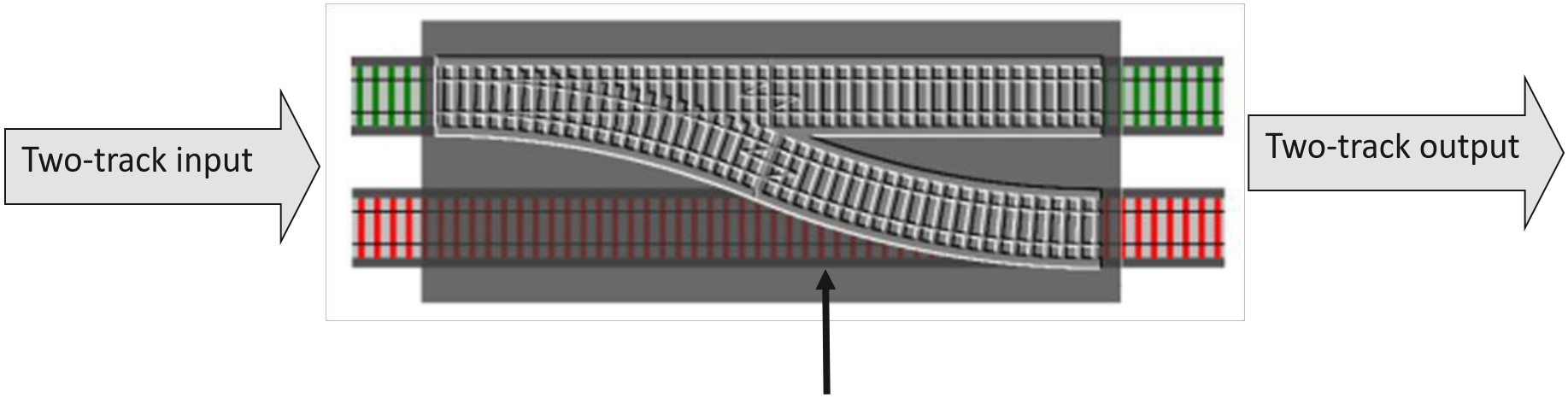


... but composing switches is not allowed!

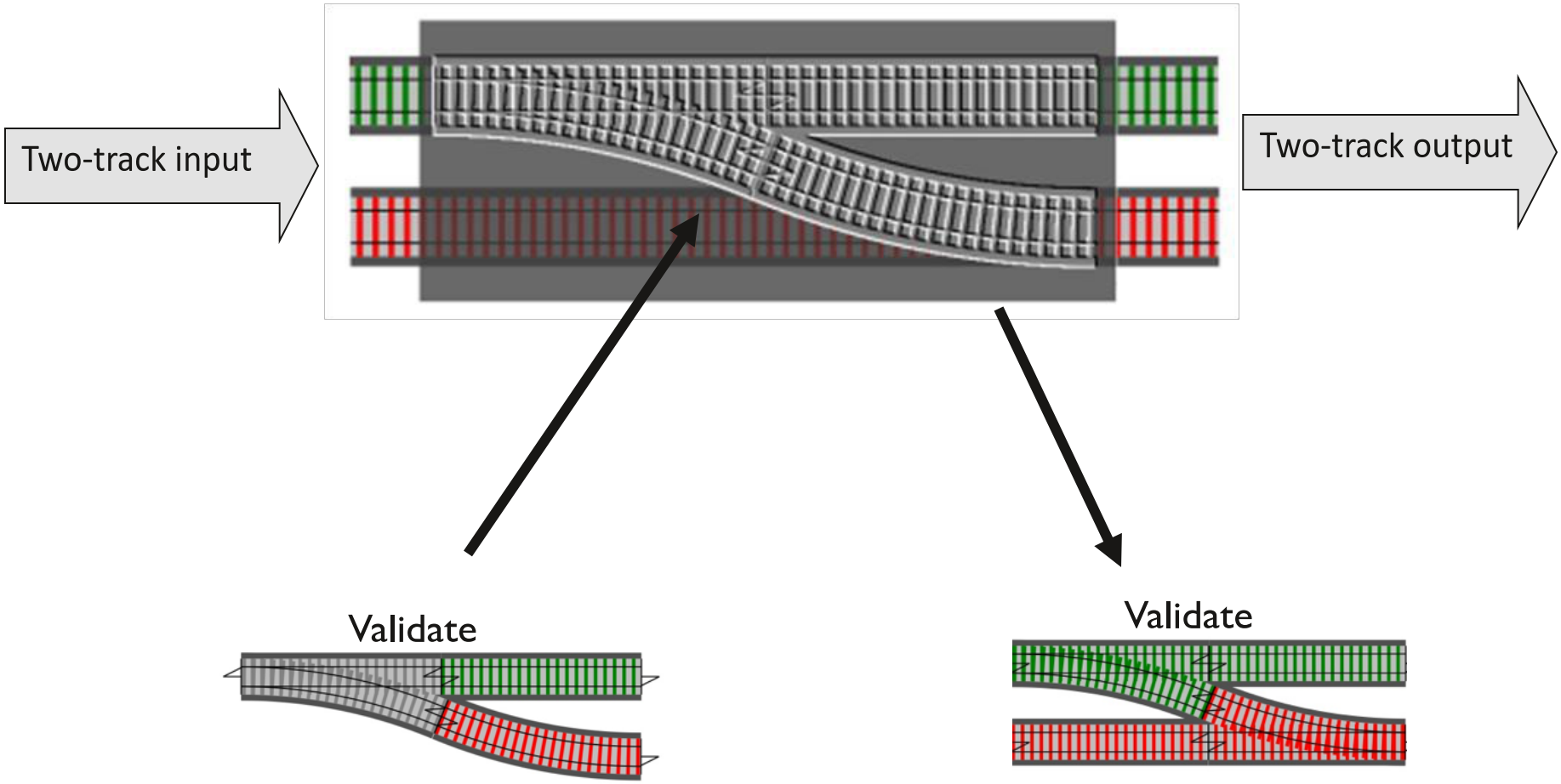
Composing switches



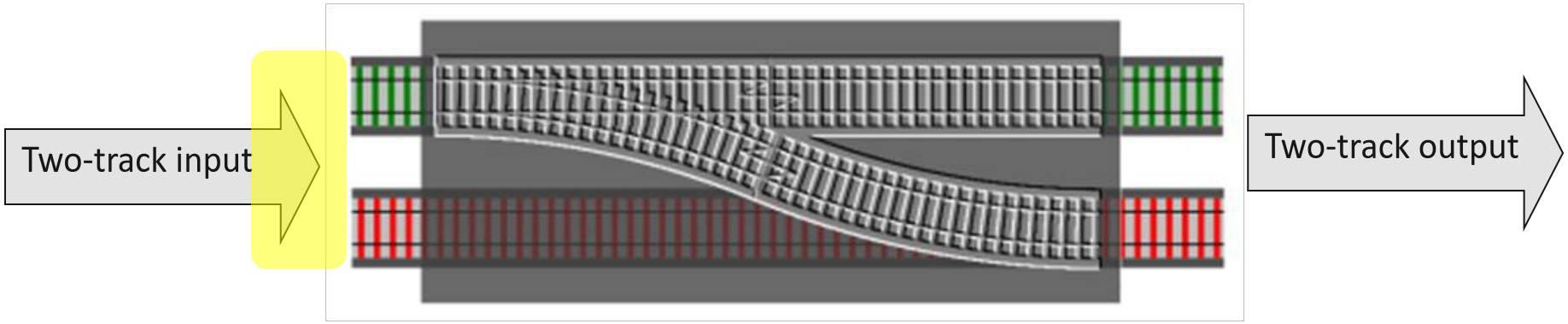
Building an adapter block



Building an adapter block

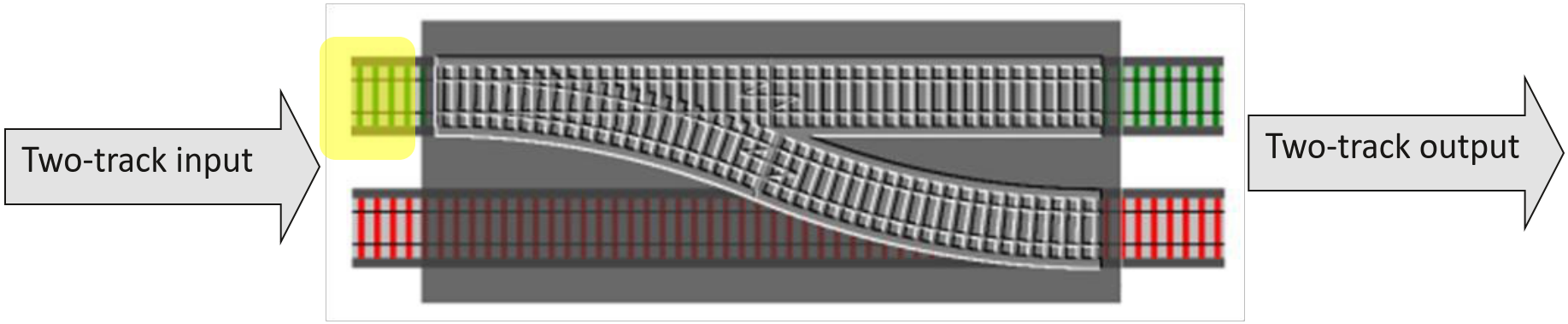


Building an adapter block



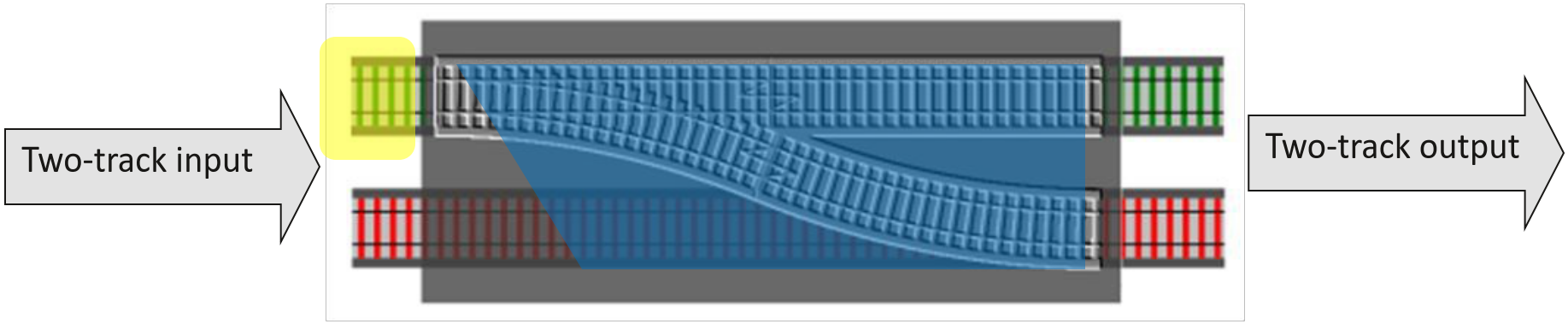

```
let adapt switchFunction =
    fun twoTrackInput ->
        match twoTrackInput with
        | Success s -> switchFunction s
        | Failure f -> Failure f
```

Building an adapter block



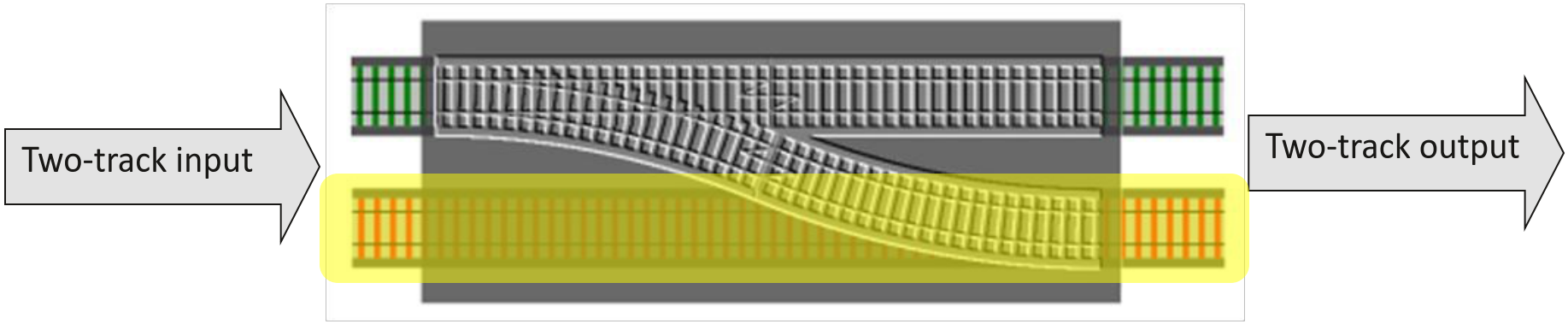

```
let adapt switchFunction =
    fun twoTrackInput ->
        match twoTrackInput with
        | Success s -> switchFunction s
        | Failure f -> Failure f
```

Building an adapter block



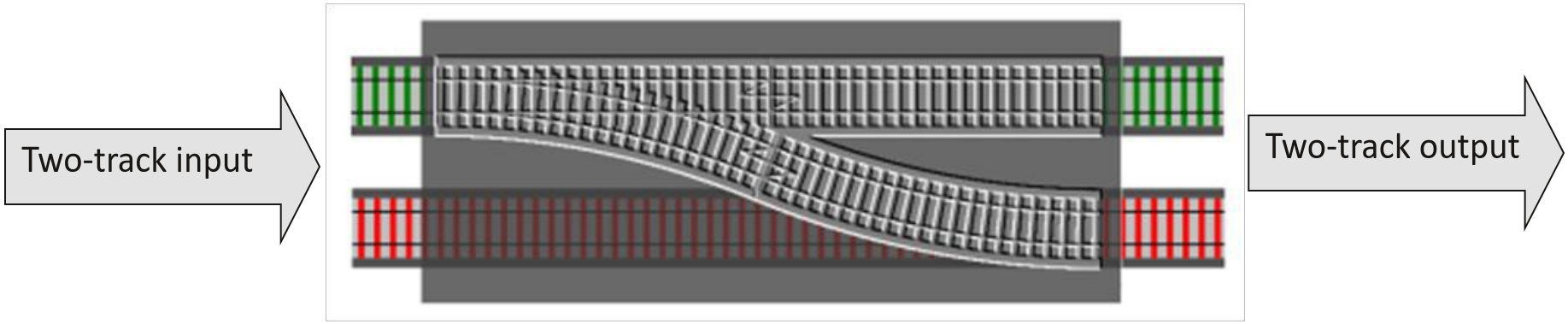

```
let adapt switchFunction =
    fun twoTrackInput ->
        match twoTrackInput with
        | Success s -> switchFunction s
        | Failure f -> Failure f
```

Building an adapter block




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

Bind as an adapter block




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

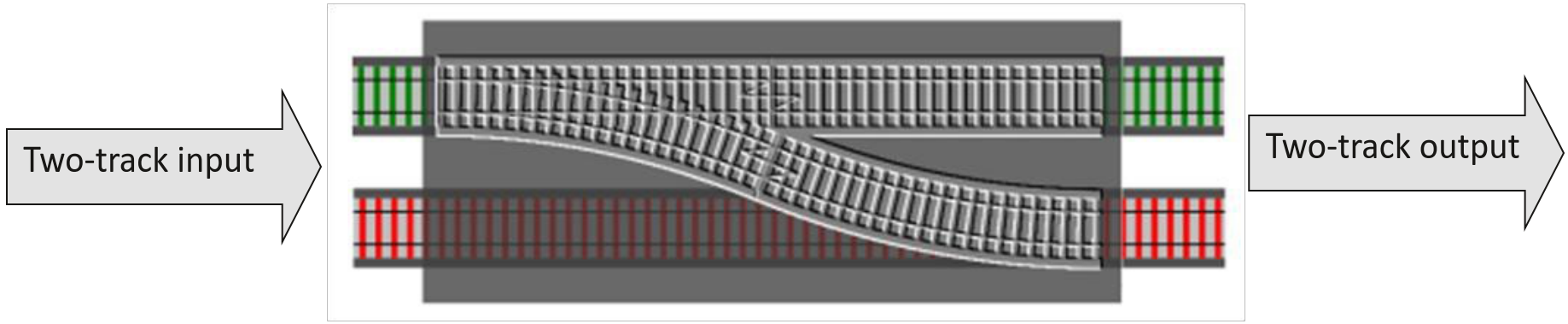
```
bind : ('a -> TwoTrack<'b>) -> TwoTrack<'a> -> TwoTrack<'b>
```

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 -> TwoTrack<'b>) -> TwoTrack<'a> -> TwoTrack<'b>
```

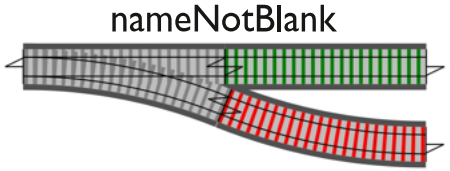
Switch function

2-track input

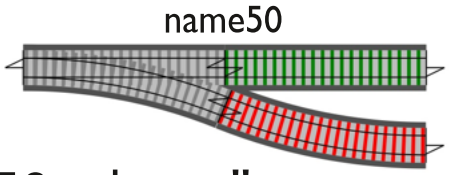
2-track output

Bind example

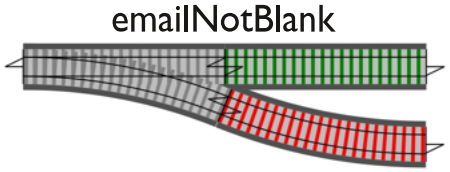
```
let nameNotBlank input =
  if input.name = "" then
     Failure "Name must not be blank"
  else Success input
```



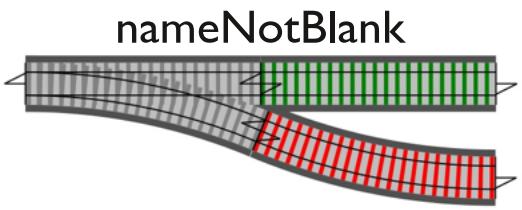

```
let name50 input =
  if input.name.Length > 50 then
     Failure "Name must not be longer than 50 chars"
  else Success input
```

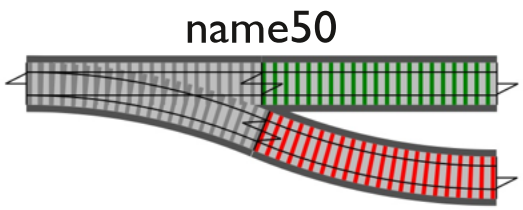


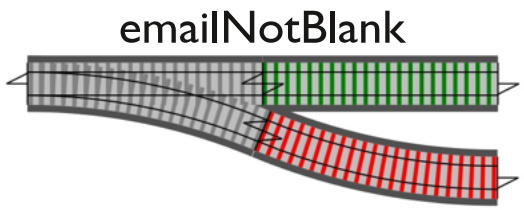

```
let emailNotBlank input =
  if input.email = "" then
     Failure "Email must not be blank"
  else Success input
```



Bind example

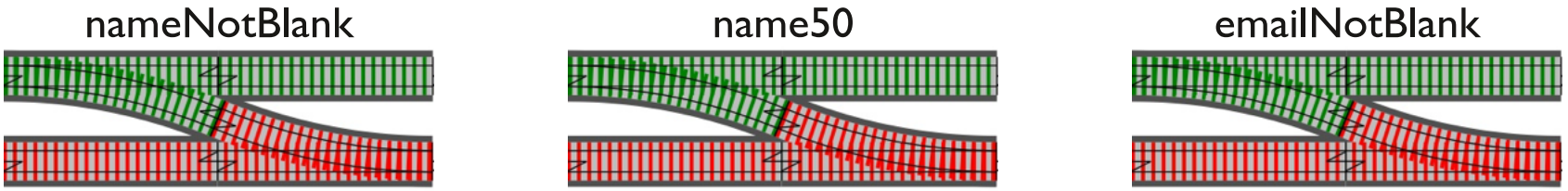







```
nameNotBlank (combined with)
name50 (combined with)
emailNotBlank
```

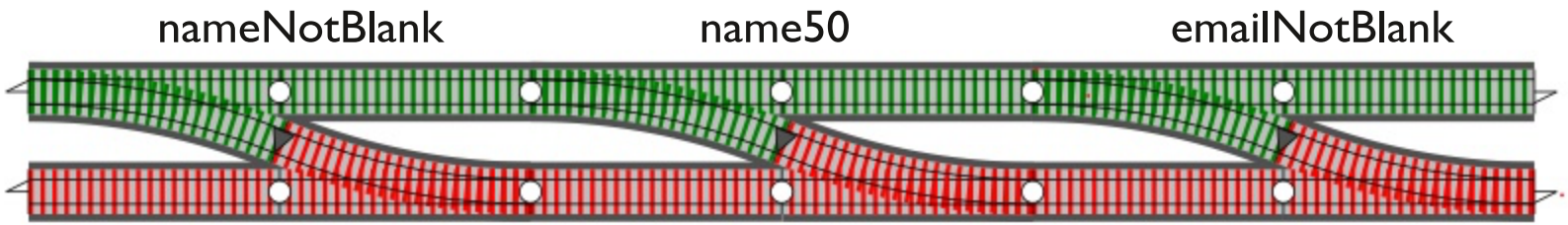
Bind example




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

use "bind" to convert to 2-track

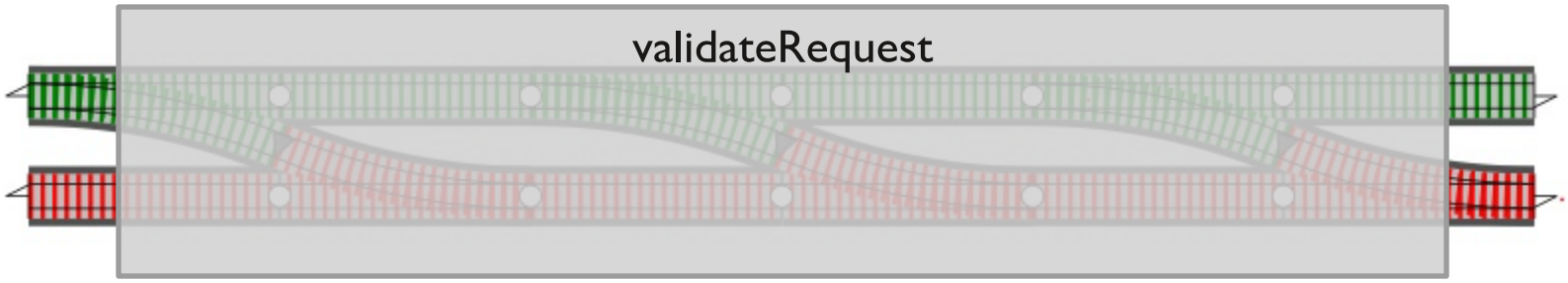
Bind example




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

then compose together

Bind example



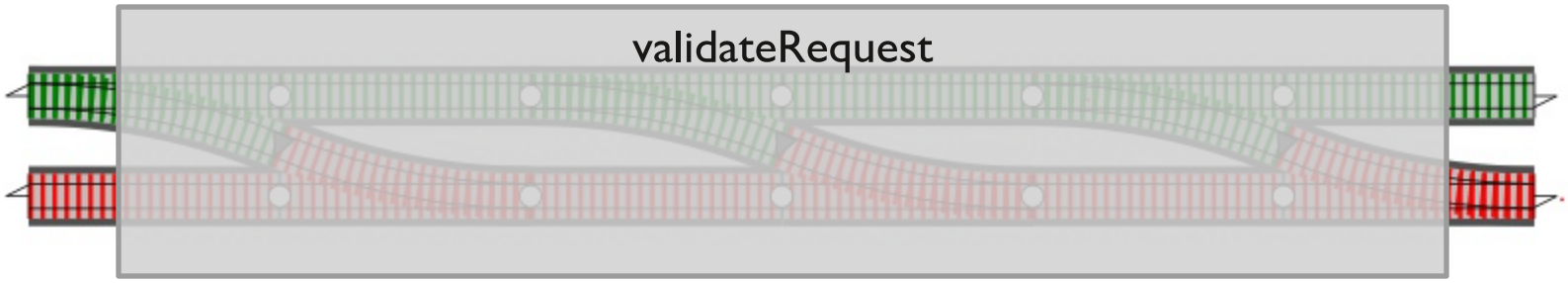
Define a function

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

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

Overall result is a new two-track function

Bind example



Common symbol for bind

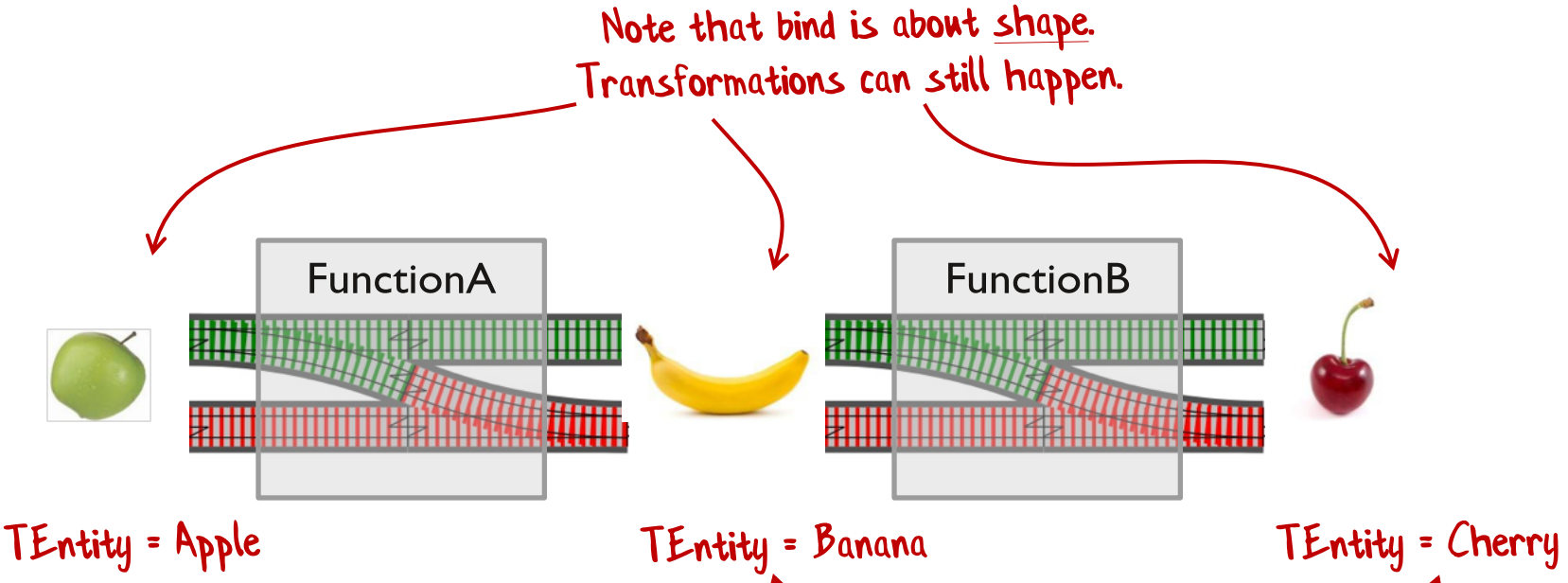
```
let (>>=) twoTrackInput switchFunction =
  bind switchFunction twoTrackInput

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

Needs a explicit parameter

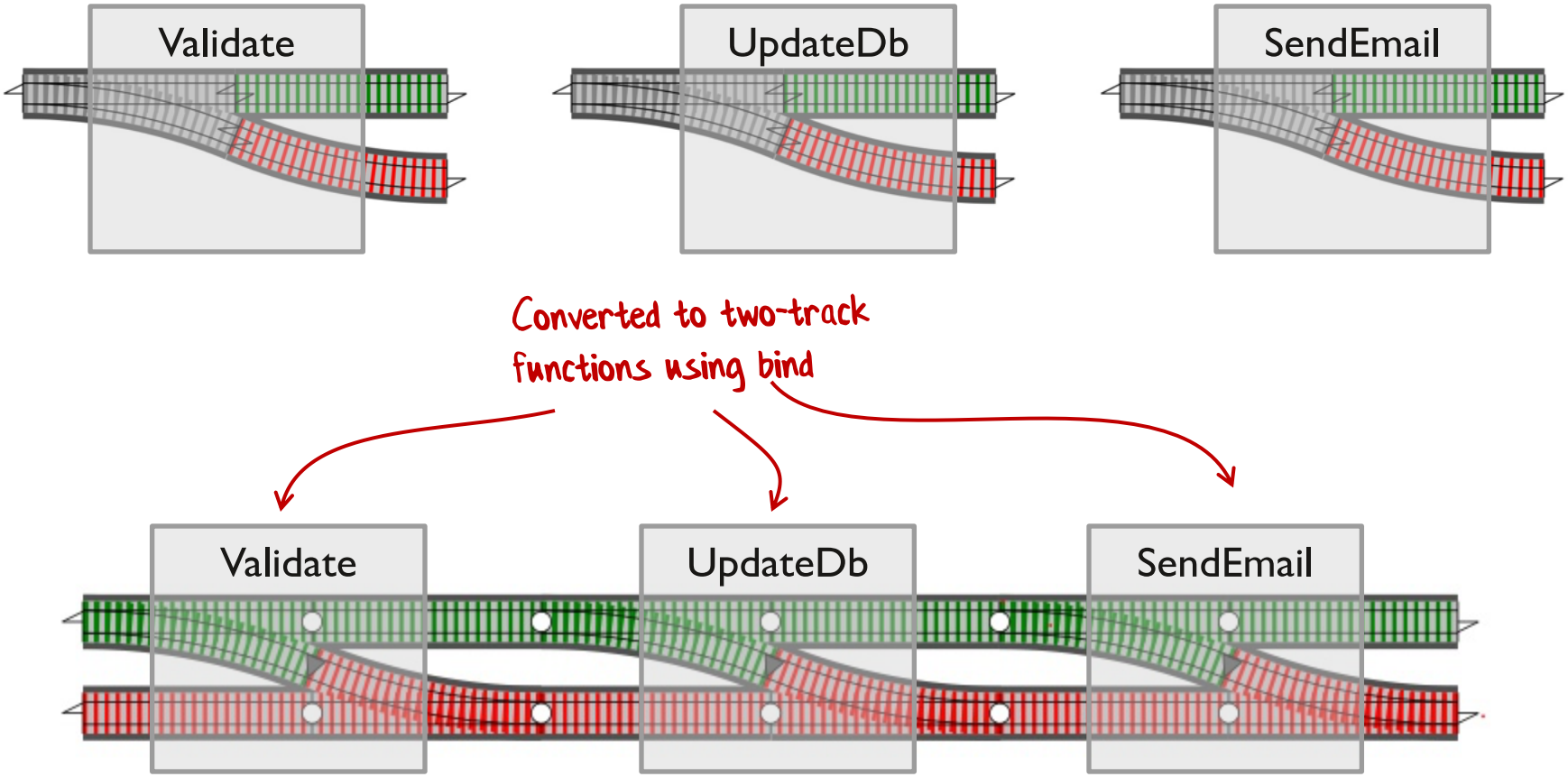
Bind symbol = F# composition symbol + railway track symbol! Coincidence?

Bind doesn't stop transformations




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

Composing switches - review



More fun with railway tracks...

Working with other functions

More fun with railway tracks...

Fitting other functions into this framework:

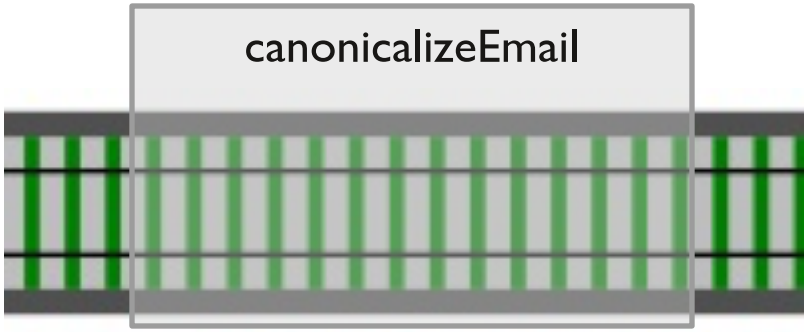
- Single track functions
- Dead-end functions
- Functions that throw exceptions
- Supervisory functions

Converting one-track functions

Fitting other functions into this framework:

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

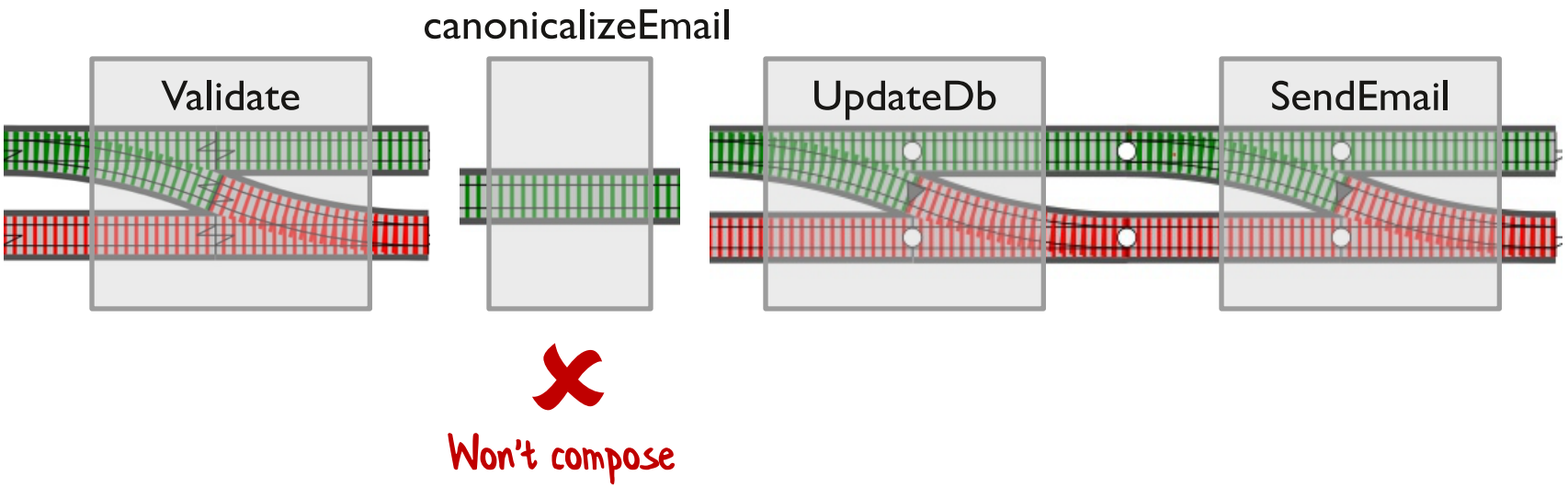
Converting one-track functions




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

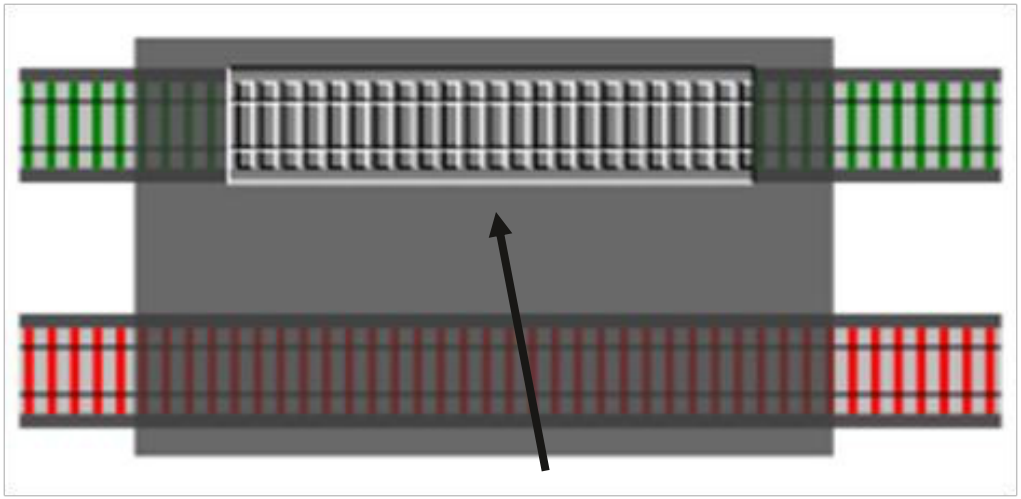
A simple function that doesn't generate errors — a "one-track" function.

Converting one-track functions



Converting one-track functions

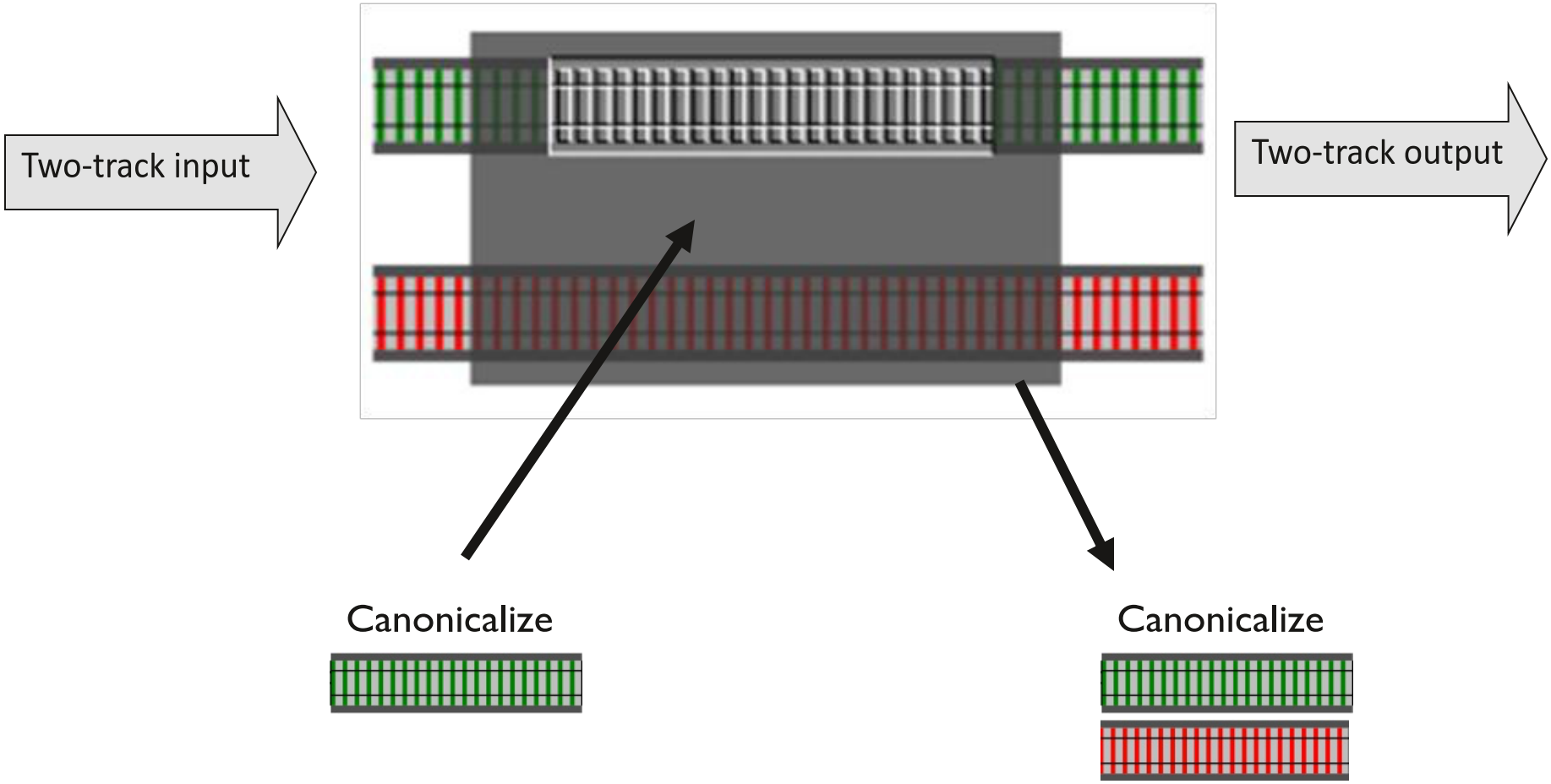
Two-track input



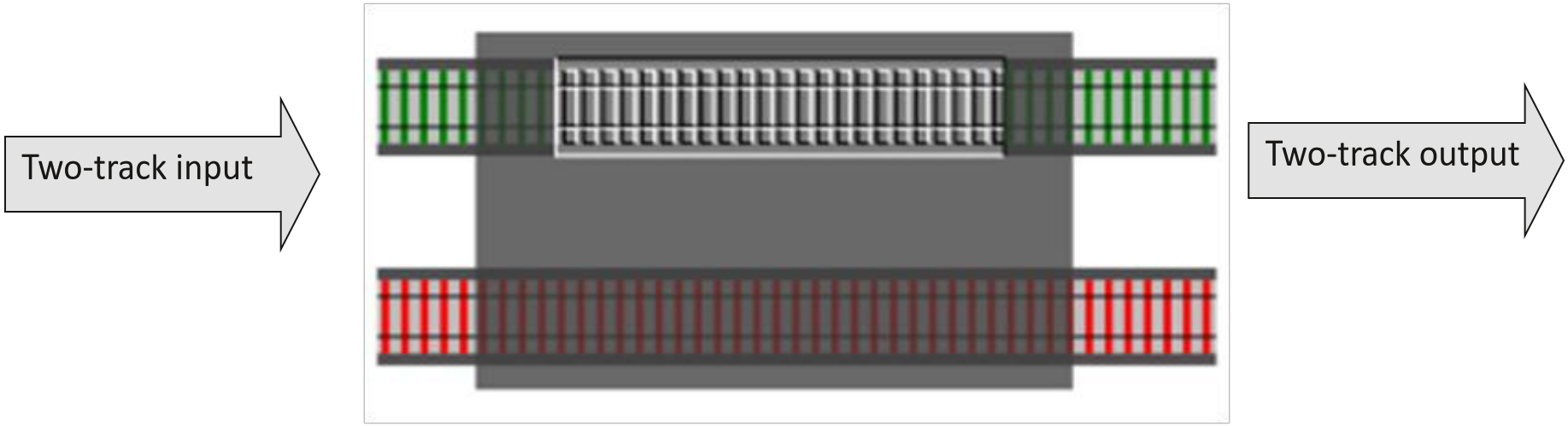
Two-track output

Slot for one-track function

Converting one-track functions



Converting one-track functions




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

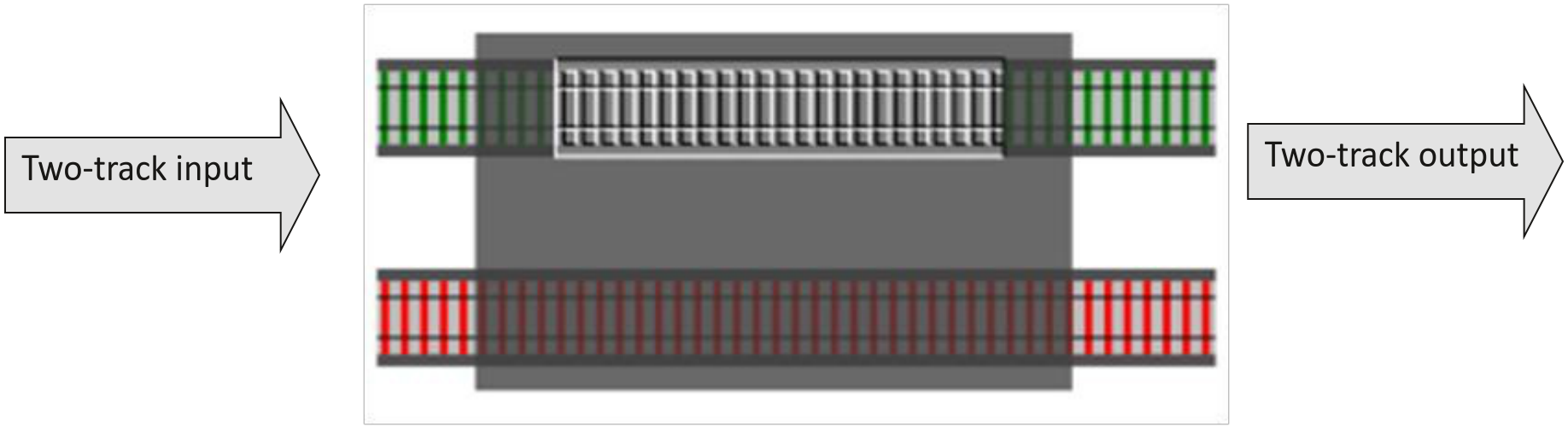
```
map : ('a -> 'b) -> TwoTrack<'a> -> TwoTrack<'b>`
```

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"

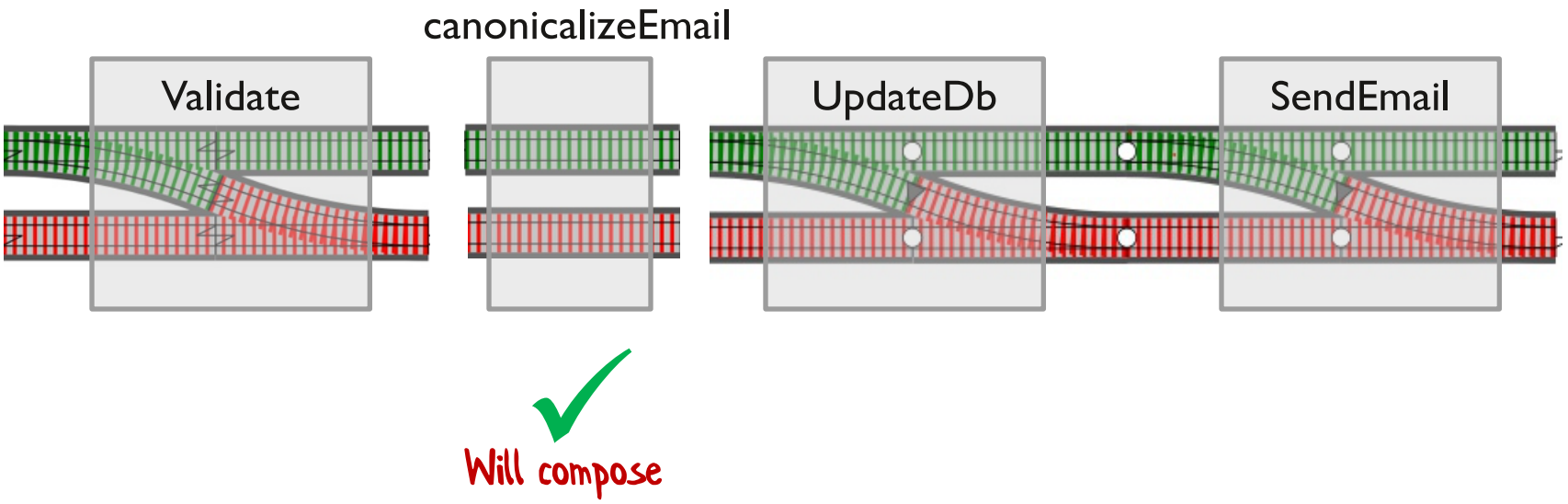
```
map : ('a -> 'b) -> TwoTrack<'a> -> TwoTrack<'b>
```

Single track function

2-track input

2-track output

Converting one-track functions

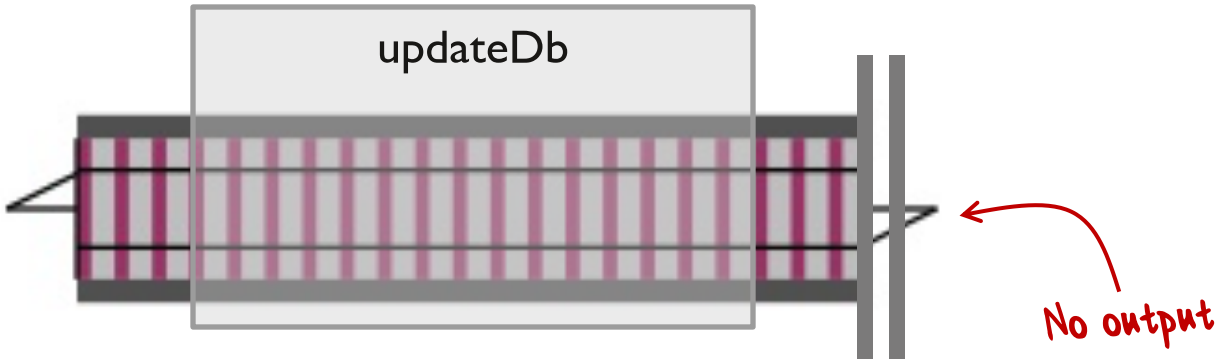


Converting dead-end functions

Fitting other functions into this framework:

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

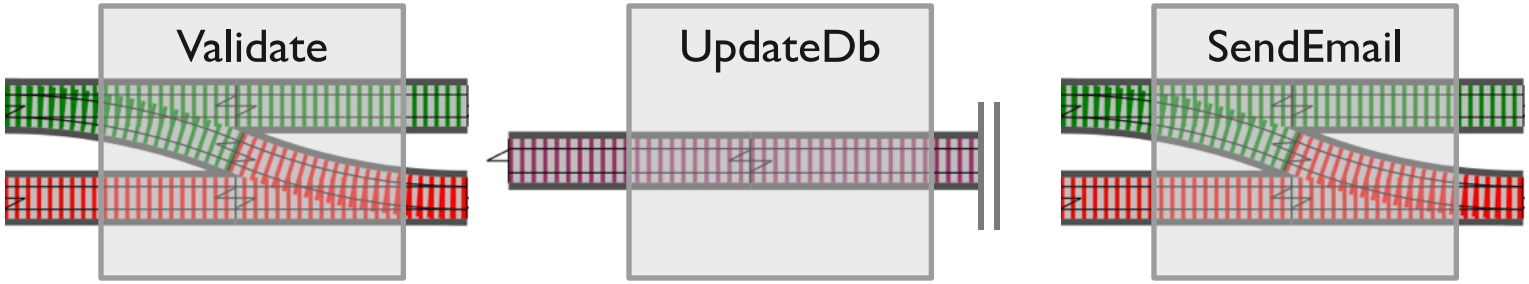
Converting dead-end functions




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

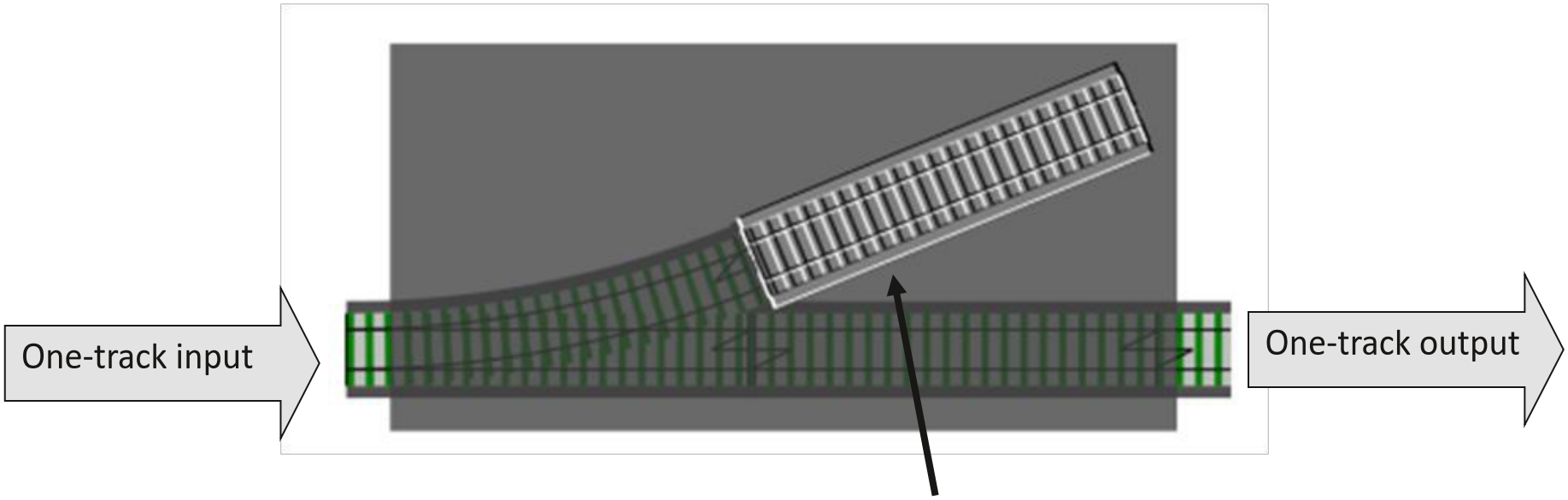
A function that doesn't return anything— a "dead-end" function.

Converting dead-end functions

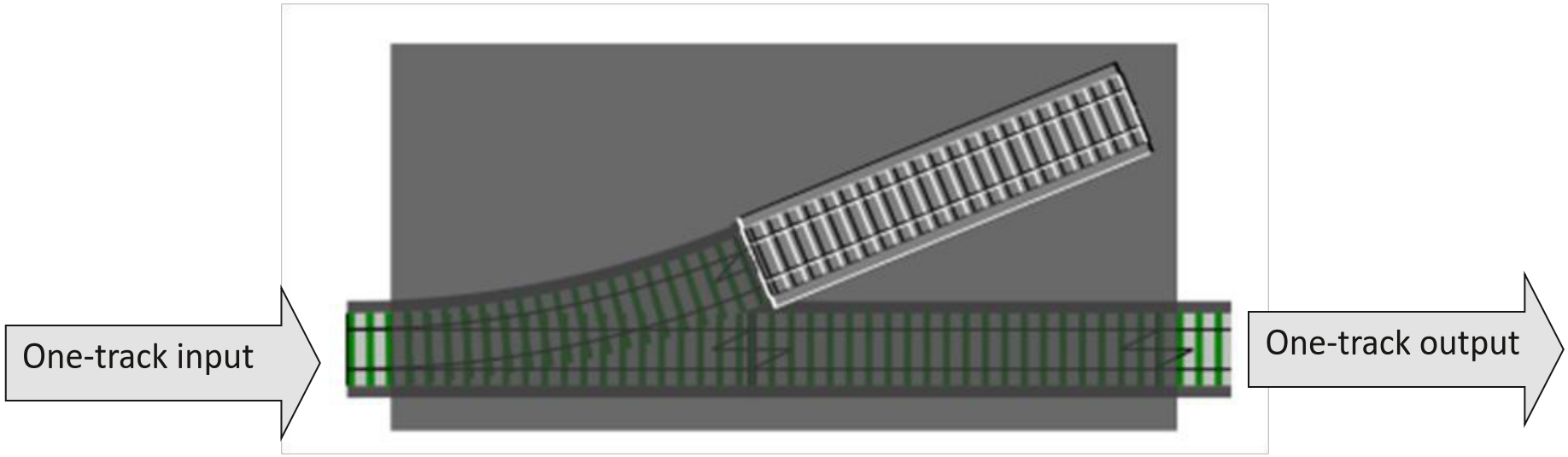


Won't compose

Converting dead-end functions



Converting dead-end functions




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

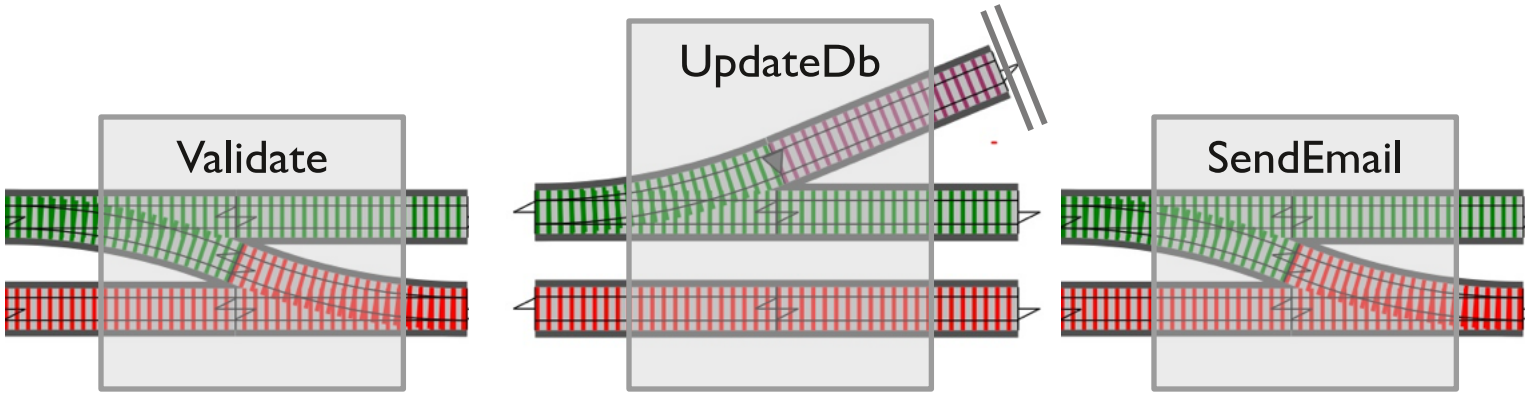
```
tee : ('a -> unit) -> 'a -> 'a
```

Dead end function

one-track input

one-track output

Converting dead-end functions



✓

Will compose

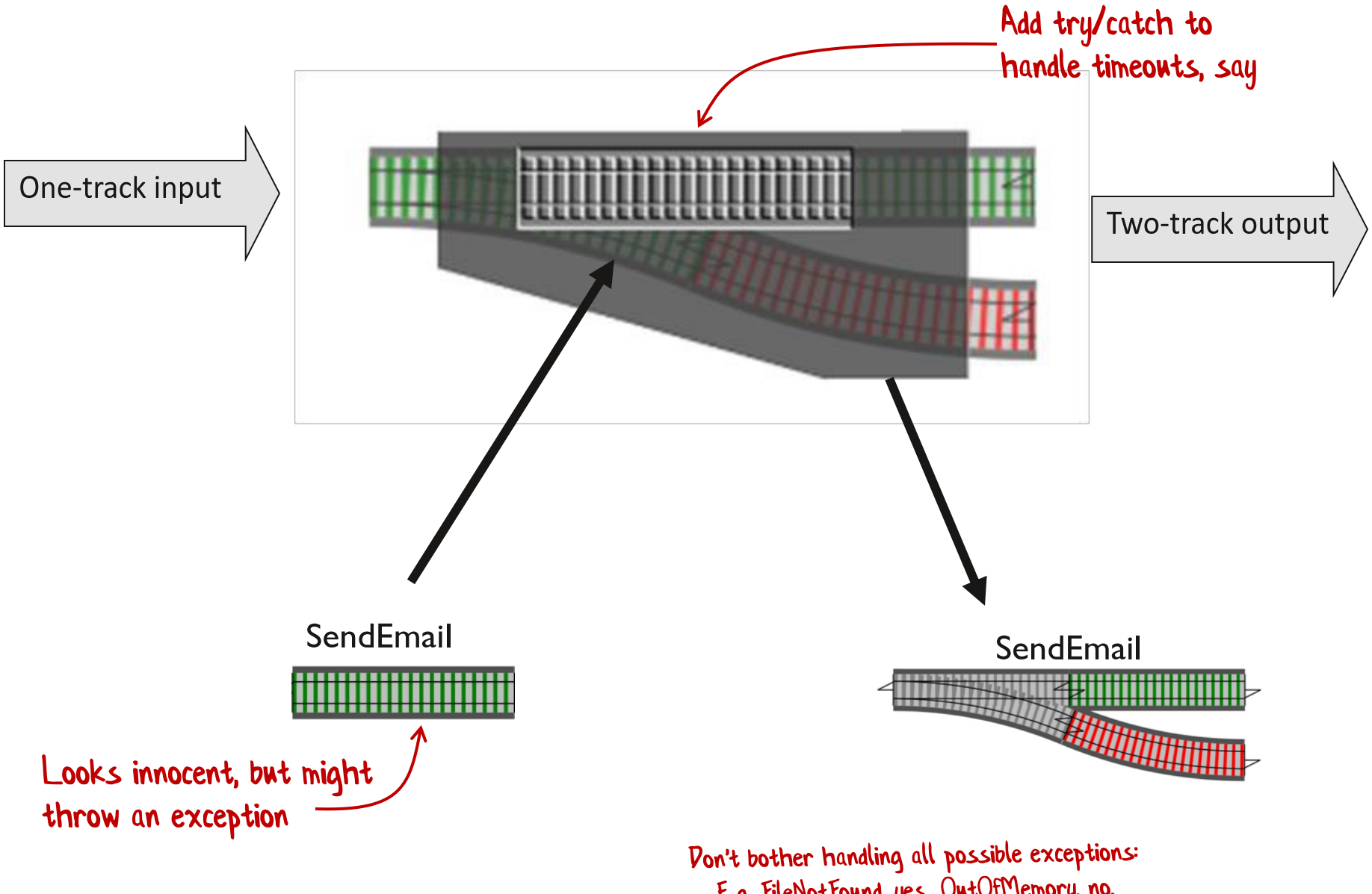
Functions that throw exceptions

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



Guideline: Convert exceptions into Failures

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

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

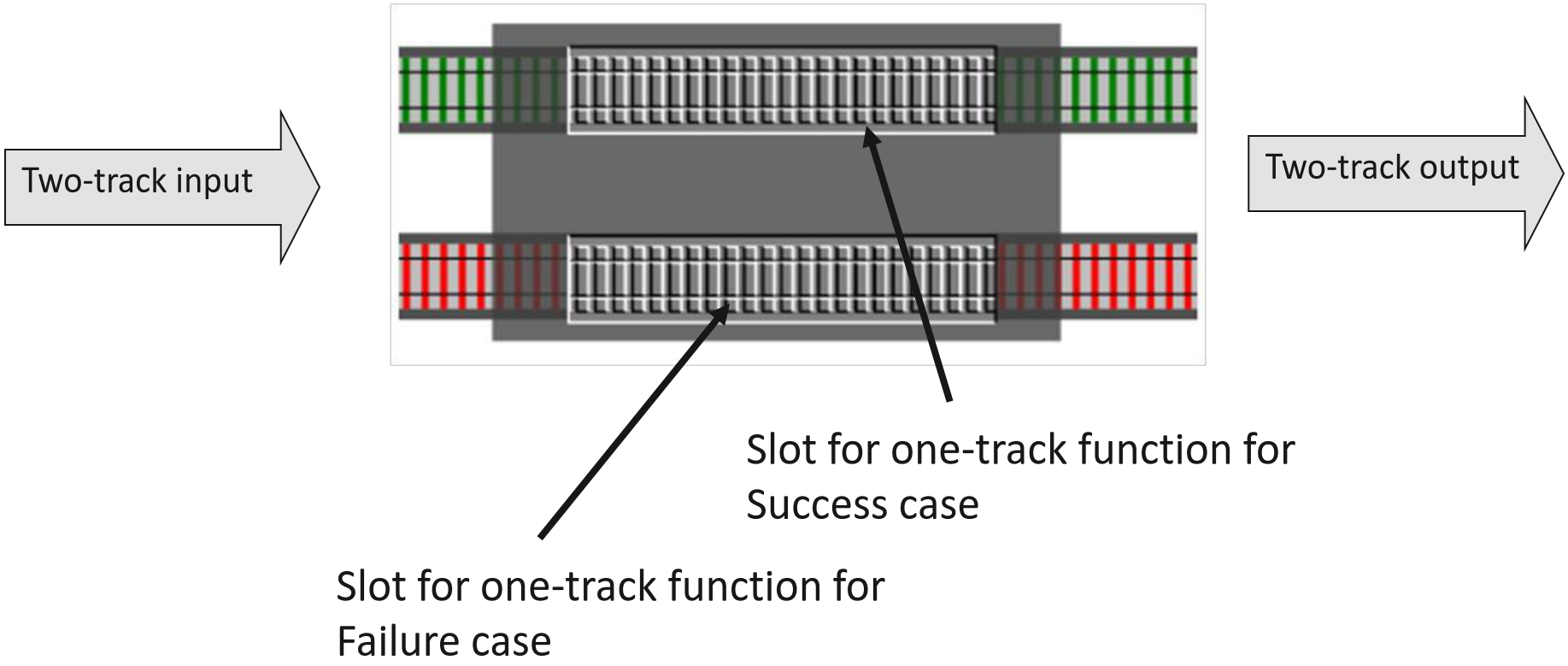
Supervisory functions

Fitting other functions into this framework:

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

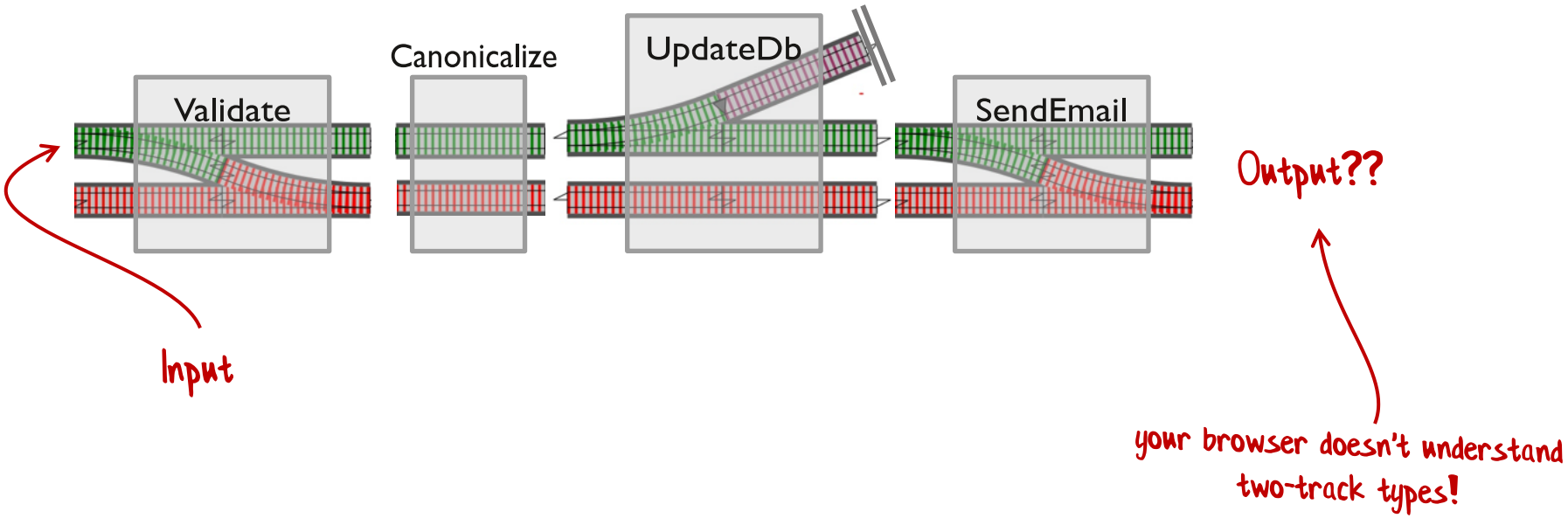
For when you need to handle *both* tracks
– e.g. tracing, logging, etc.

Supervisory functions

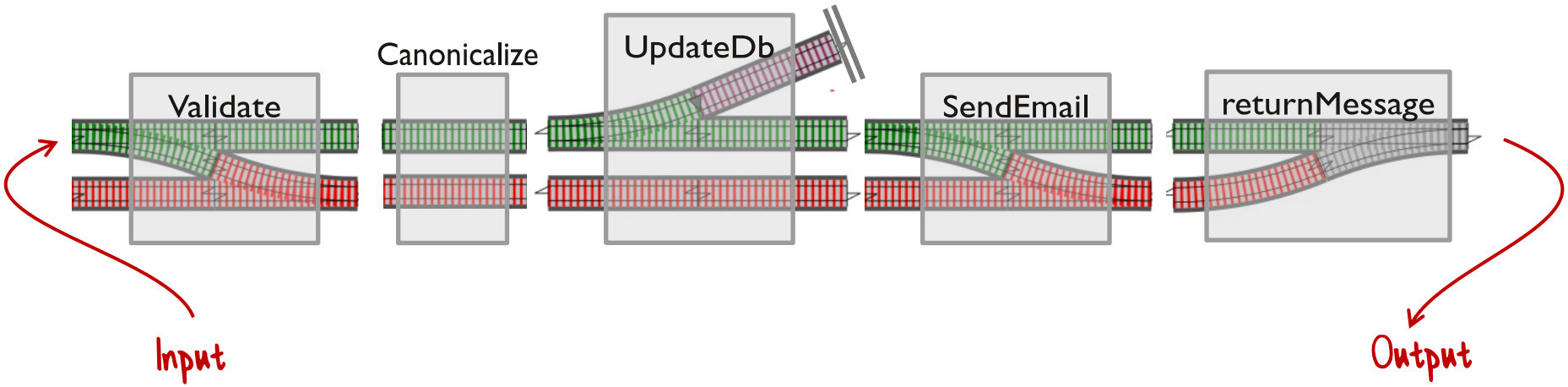


Putting it all together

Putting it all together



Putting it all together




```
let returnMessage result =
  match result with
  | Success obj -> OK obj.ToJson()
  | Failure msg -> BadRequest msg
```

Putting it all together - review

You can fit most functions into this model.

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

Not a solution for everything, but a good starting point.

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 updateCustomer =
  receiveRequest
  >> validateRequest
  >> updateDbFromRequest
  >> sendEmail
  >> returnMessage
```

Before – without error handling

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

After – with error handling

Still clean and elegant

Extending the framework:

- Designing for errors
- Parallel tracks
- Domain events

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 TwoTrack<'TEntity> =
   | Success of 'TEntity
   | Failure of string
```

Using strings is not good

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 TwoTrack<'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!

Designing for errors

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

Documentation of everything that can go wrong --

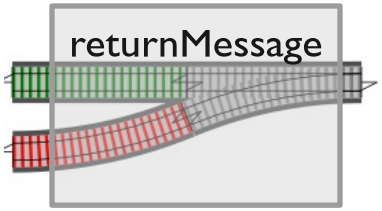
And it's type-safe documentation that can't go out of date!

As we develop the code, we can build up a complete list of everything that could go wrong

one single place to see all the error cases

Also triggers important DDD conversations

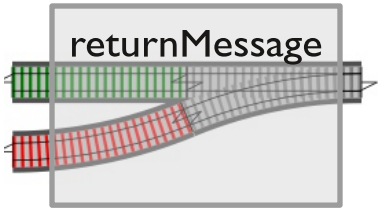
Designing for errors – converting to strings




```
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)

Designing for errors - review

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

Documentation of everything that can go wrong.

Type-safe -- can't go out of date!

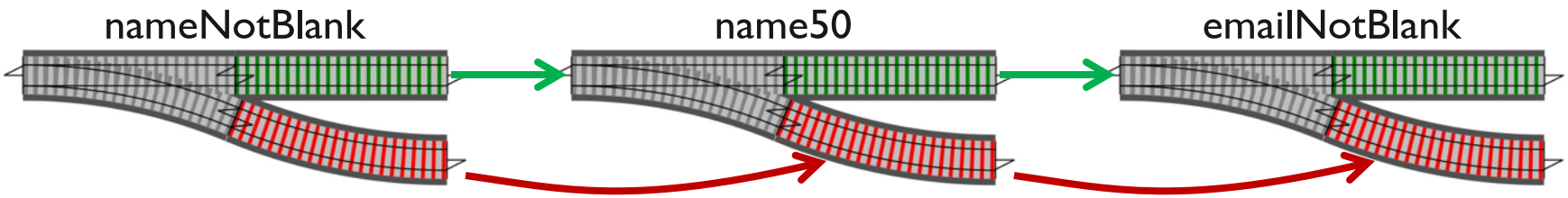
Surfaces hidden requirements.

Test against error codes, not strings.

Makes translation easier.

Parallel tracks

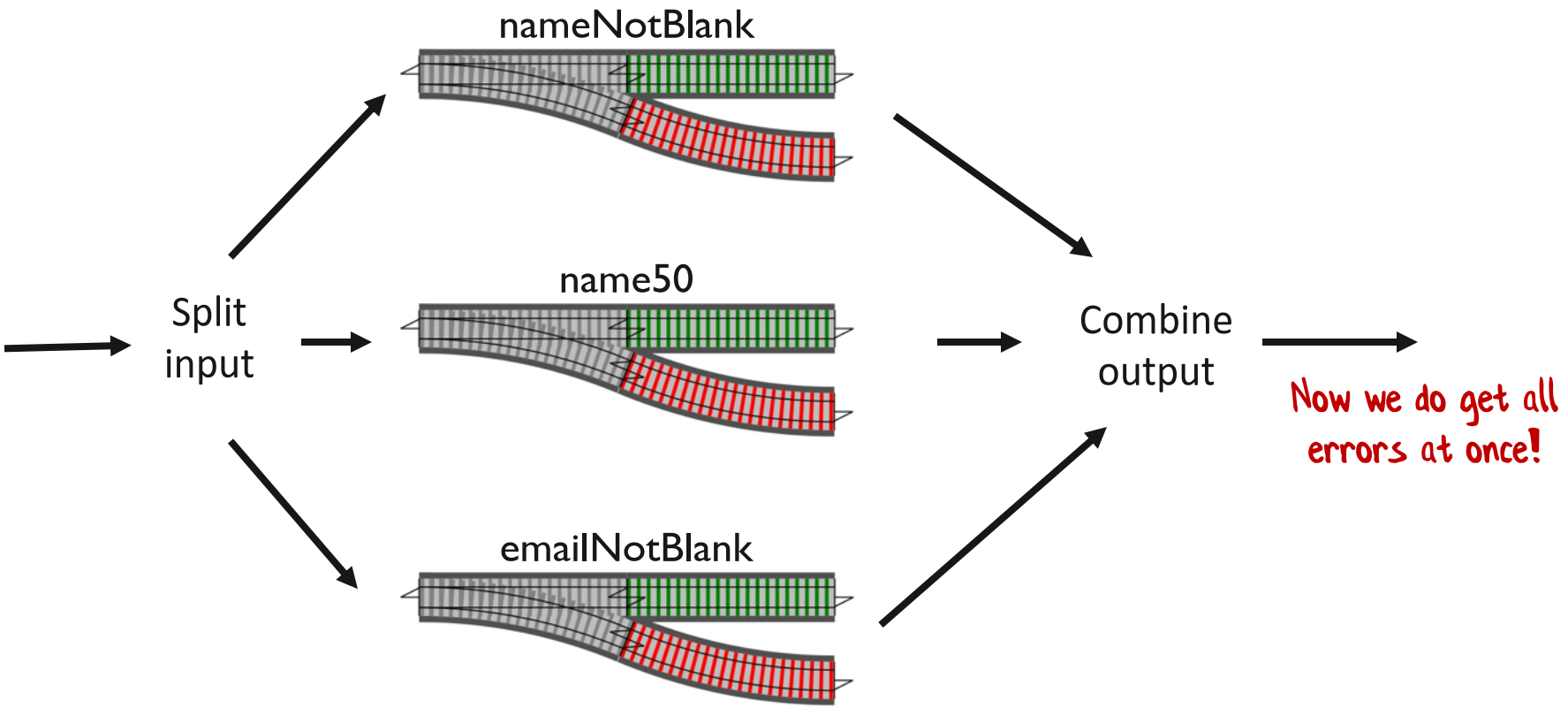
Parallel validation



Problem: Validation done in series.
So only one error at a time is returned

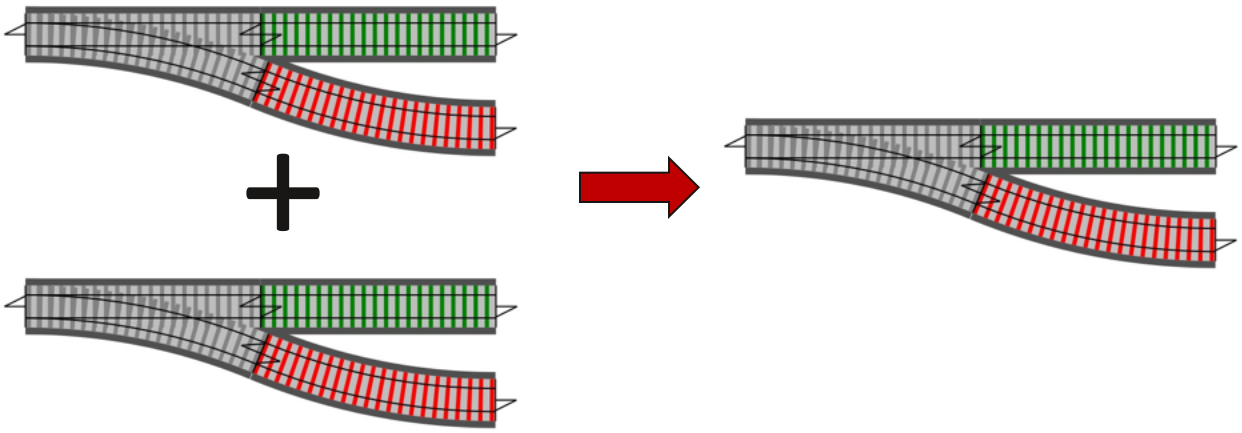
It would be nice to return all validation errors at once.

Parallel validation



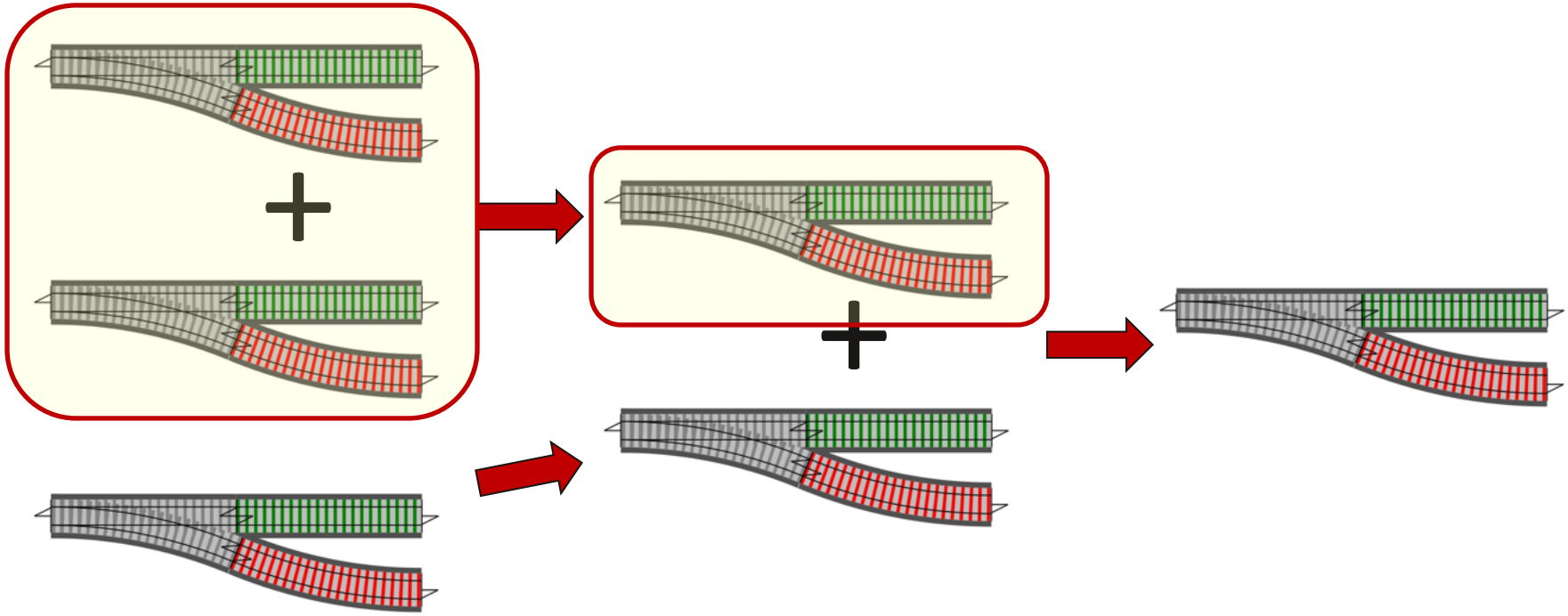
Combining switches

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



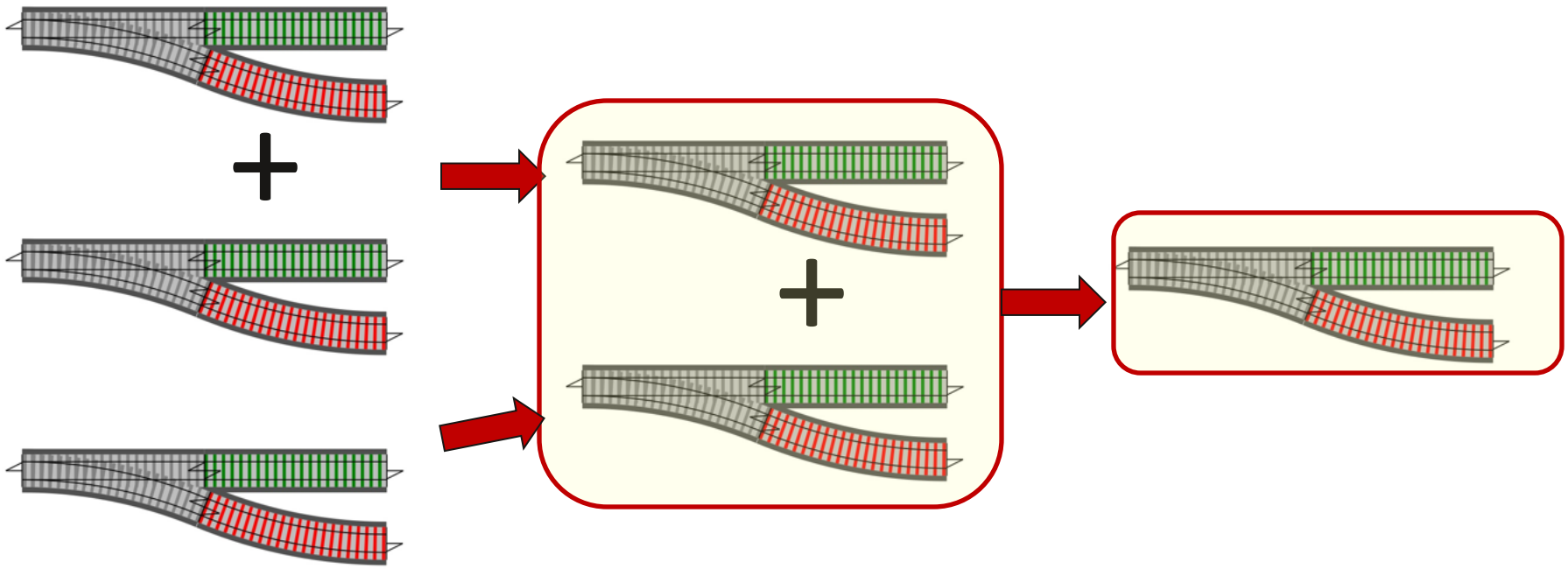
Combining switches

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



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.

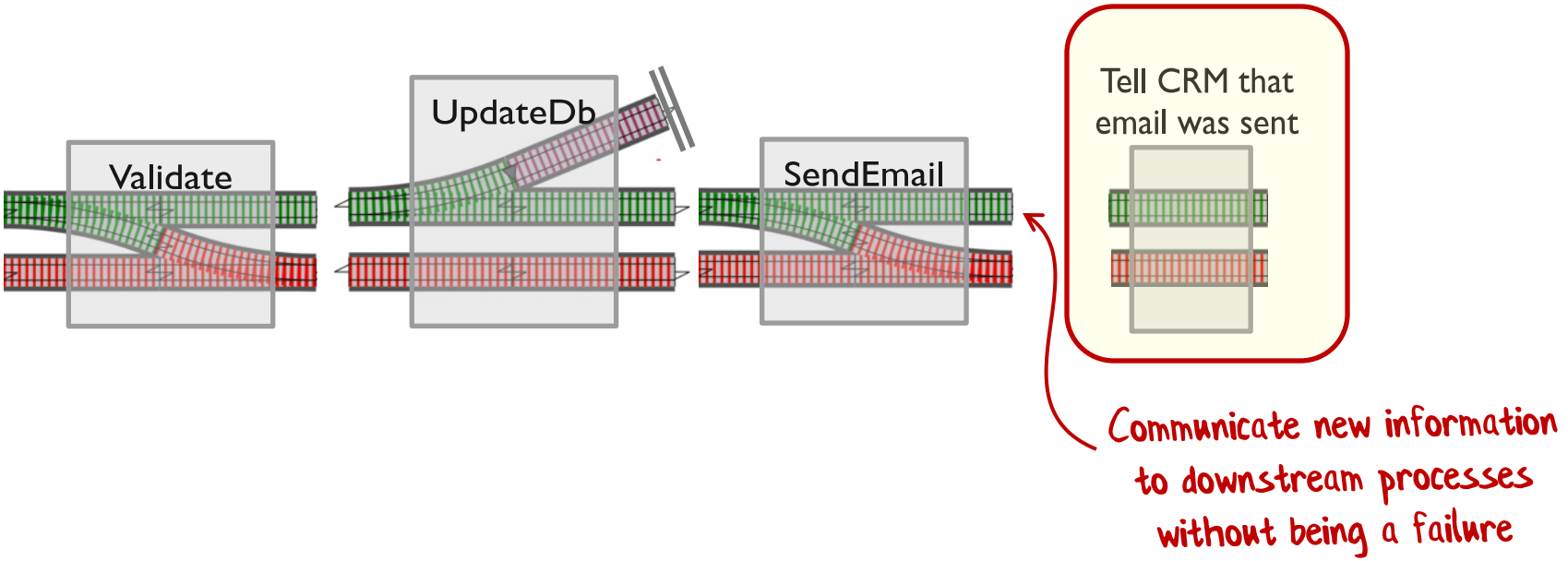


-> For more, see "monoids without tears"

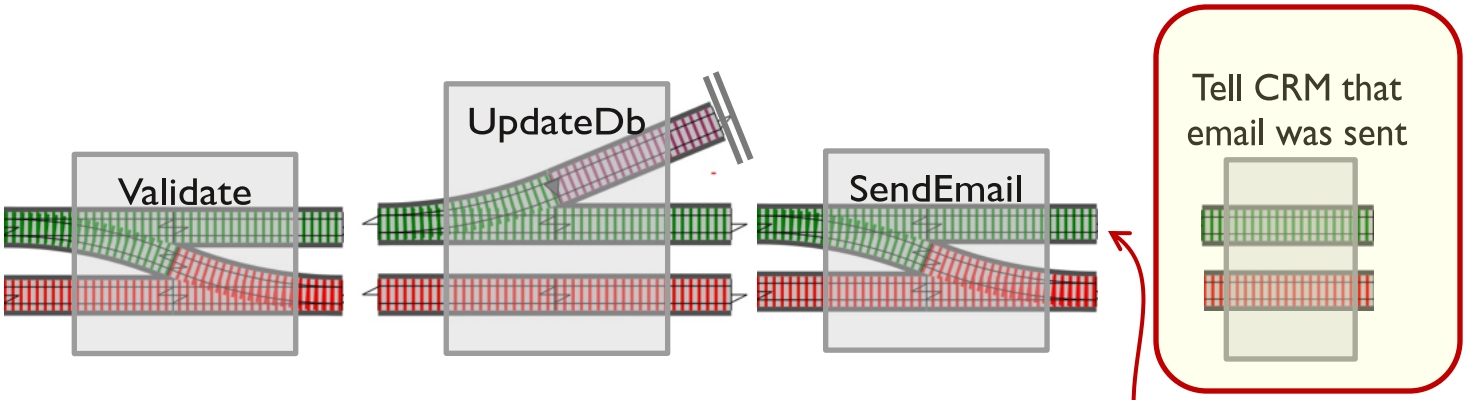
Domain events

Communicating information to downstream functions

Events are not errors



Events are not errors

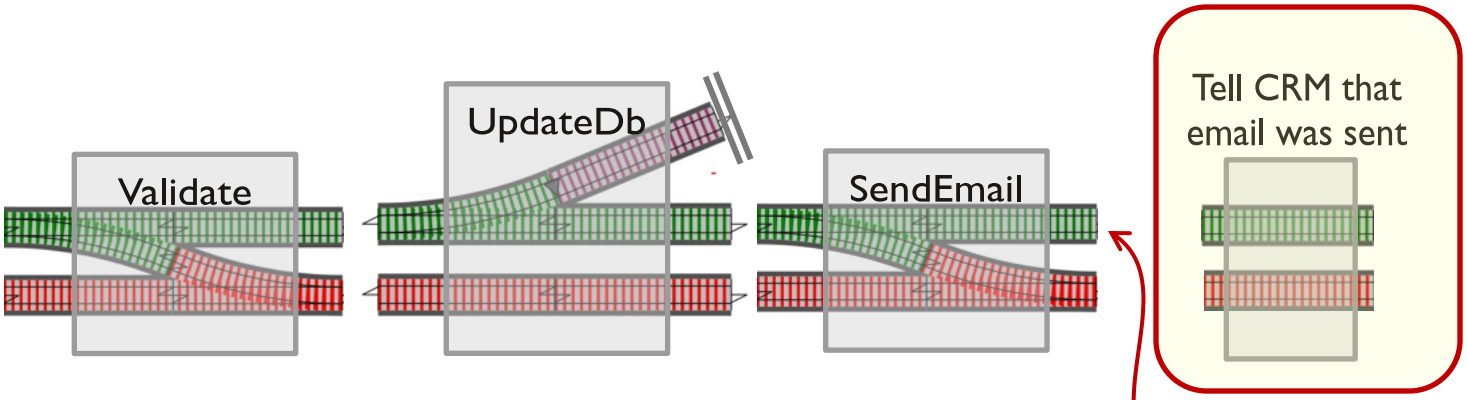



```
type MyUseCaseMessage =
  | NameMustNotBeBlank
  | EmailMustNotBeBlank
  | EmailNotValid of EmailAddress
  // database errors
  | UserIdNotValid of UserId
  // SMTP errors
  | SmtpTimeout of SmtpConnection
  // Domain events
  | UserSaved of AuditInfo
  | EmailSent of EmailAddress * MsgId
```

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

Some topics not covered...

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

Topics not covered

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

Summary

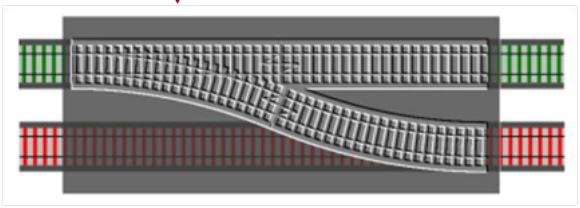
A recipe for handling errors in a functional way

Recipe for handling errors in a functional way

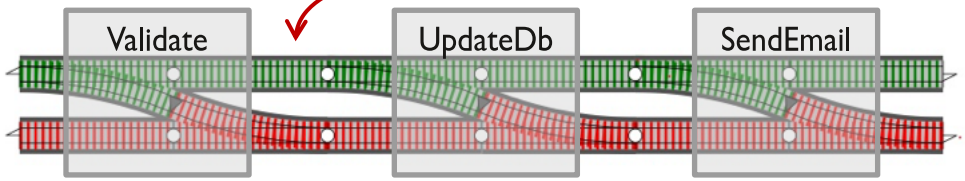
Step 1: Create a Result type

```
type TwoTrack<'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
```