Domain Modeling with Types

What can we do with
an algebraic type system?

AND becomes a record type

```
data Name =
  FirstName
  AND MiddleInitial
  AND LastName
```

```
type Name= {
  FirstName: string
  MiddleInitial: string
  LastName: string
}
```

Use curly braces

Lists are built-in

```
data Order =
  OrderId
  AND list of OrderLines
```

```
type Order = {
  OrderId : OrderId
  OrderLines : OrderLine list
}
```

OR becomes a choice type

```
data PaymentMethod =
  Cash
  OR Card (with CardInfo)
  OR PayPal (with EmailAddress)
```

```
type PaymentMethod =
  | Cash
  | Card of CardInfo
  | PayPal of EmailAddress
```

Use vertical bar for choices

Workflows become function types

```
Workflow: "Place order"
  primary input:
    An order form
  output events:
    "Order Placed" event
```

```
type PlaceOrder =
  OrderForm -> OrderPlaced
```

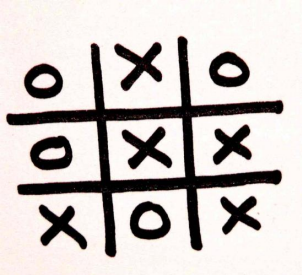
Use arrows to separate input from output

Exercise:

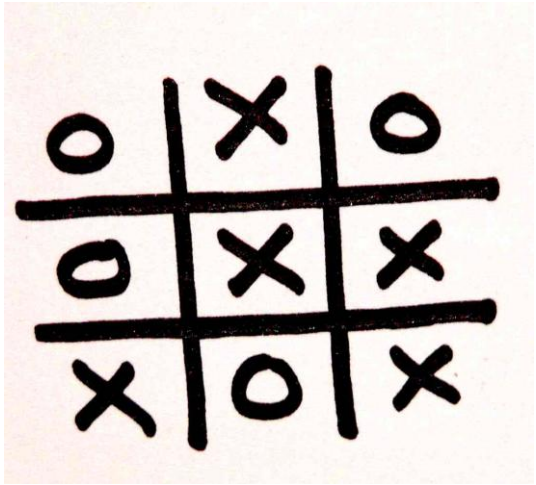
`00a-PlayWithTypes.fsx`

Also, convert this morning's domain models to types

`See gdocs for link to text`



Convert the domain models to types



A domain modeling challenge

```
type Contact = {

  FirstName: string
  MiddleInitial: string
  LastName: string

  EmailAddress: string
  IsEmailVerified: bool
  }     // true if ownership of
        // email address is confirmed
```

How many things are wrong with this design?

```
type Contact = {

  FirstName: string
  MiddleInitial: string
  LastName: string

  EmailAddress: string
  IsEmailVerified: bool
  }
```

Which values are optional?

```
type Contact = {
    FirstName: string
    MiddleInitial: string
    LastName: string

    EmailAddress: string
    IsEmailVerified: bool
    }
```

Must not be more than 50 chars

What are the constraints?

```
type Contact = {
    FirstName: string
    MiddleInitial: string
    LastName: string

    EmailAddress: string
    IsEmailVerified: bool
    }
```

Must be updated as a group

Which fields are linked?

```
type Contact = {

  FirstName: string
  MiddleInitial: string
  LastName: string

  EmailAddress: string
  IsEmailVerified: bool
  }
```

Must be reset if email is changed

What is the domain logic?

```
type Contact = {

  FirstName: string
  MiddleInitial: string
  LastName: string

  EmailAddress: string
  IsEmailVerified: bool
  }
```

Which values are optional?

What are the constraints?

Which fields are linked?

Any domain logic?

We can model all these things with types!

Modeling optional values

Required vs. Optional

```
type PersonalName =
    {
    FirstName: string;
    MiddleInitial: string;
    LastName: string;
    }
```

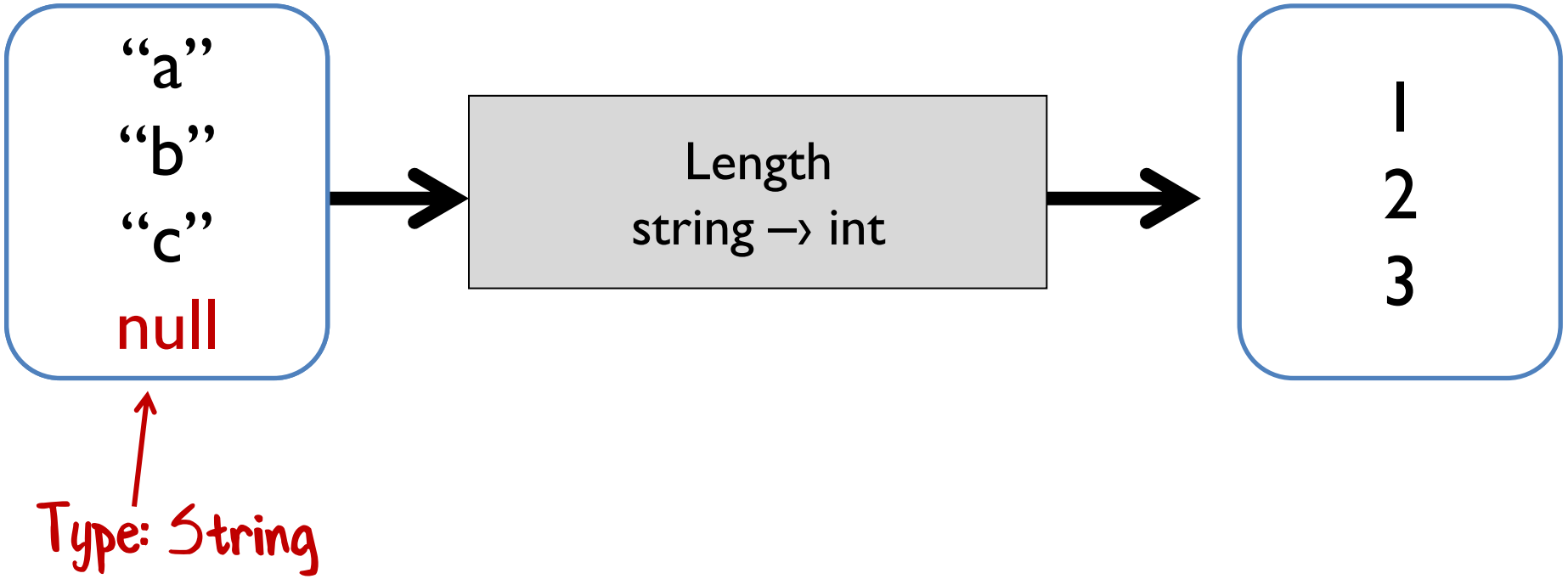
required

optional

required

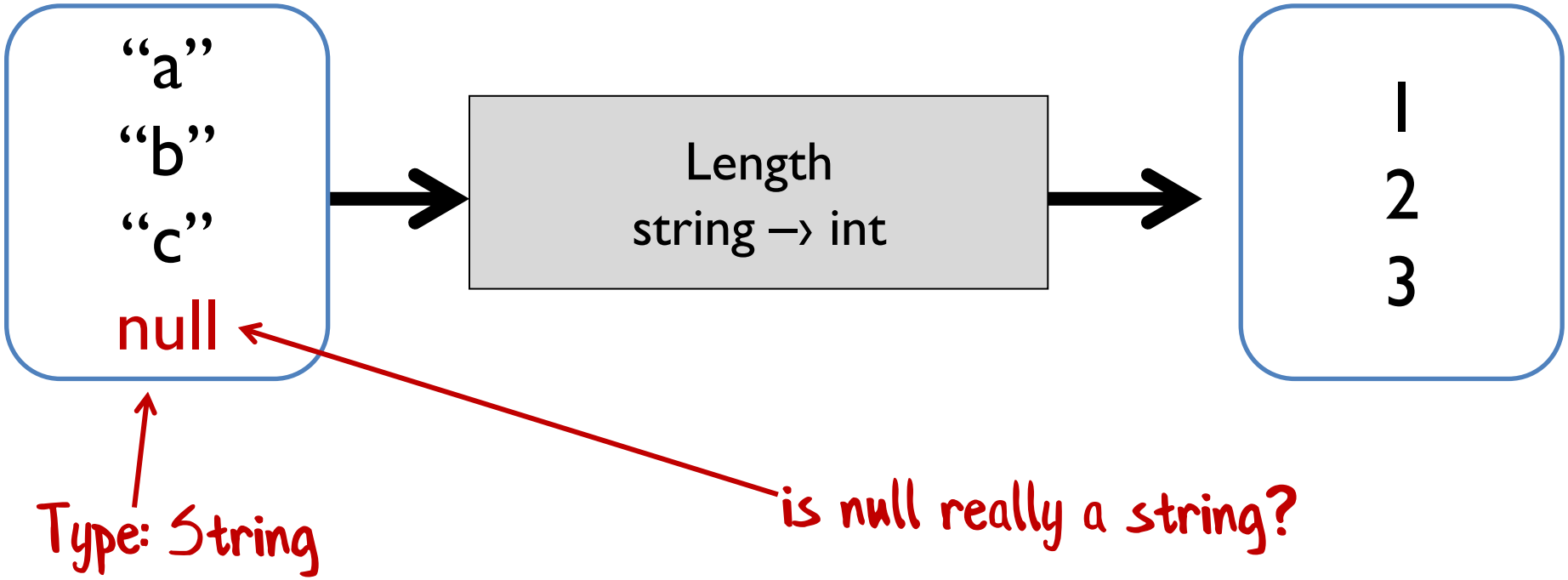
How can we represent optional values?

Null is not the same as “optional”



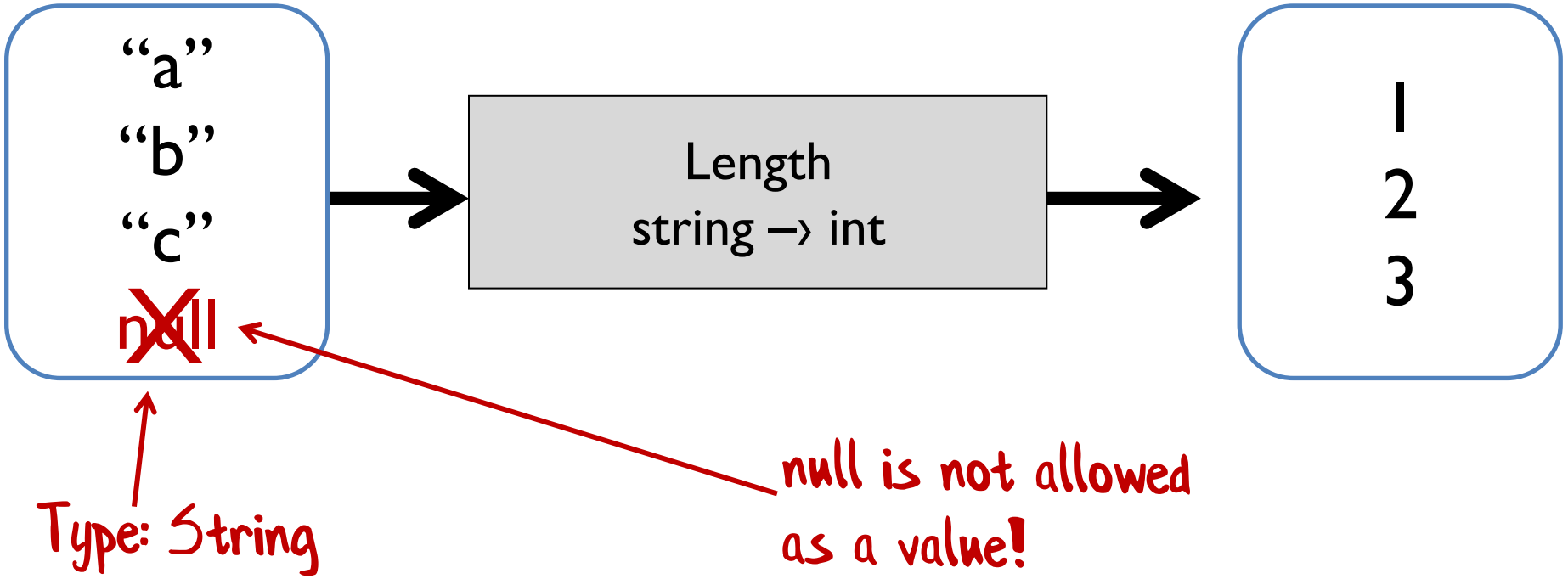

Spock, set phasers to null!
That is illogical, Captain

Null is not the same as “optional”

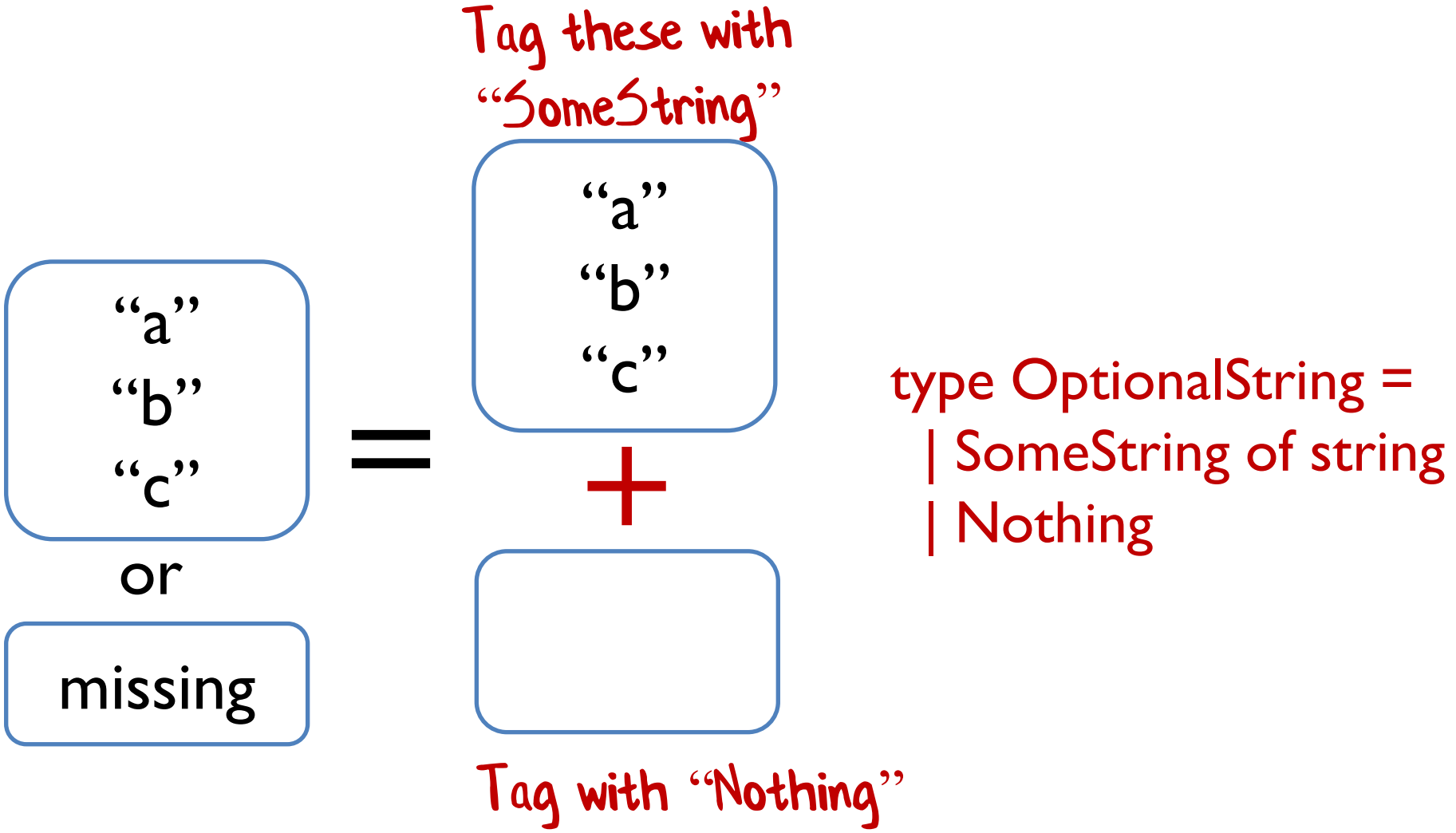


"null is the Saruman of static typing"

Null is not allowed



A better way for optional values



Defining optional types

```
type OptionalString =
    | SomeString of string
    | Nothing

type OptionalInt =
    | SomeInt of int
    | Nothing

type OptionalBool =
    | SomeBool of bool
    | Nothing
```

Duplicate code?

The built-in “Option” type

```
type Option<'T> =
    | Some of 'T
    | None
```

generic type

```
type PersonalName =
    {
    FirstName: string
    MiddleInitial: string
    LastName: string
    }
```

The built-in “Option” type

```
type Option<'T> =
    | Some of 'T
    | None

type PersonalName =
    {
    FirstName: string
    MiddleInitial: Option<string>
    LastName: string
    }
```

The built-in “Option” type

```
type Option<'T> =
    | Some of 'T
    | None

type PersonalName =
    {
    FirstName: string
    MiddleInitial: string option
    LastName: string
    }
```

nice and readable!

Modeling simple values and constrained values

Modeling simple values and constrained values

- Simple values should not be represented by primitive types
  - Emails are not strings
  - CustomerIds are not ints
- Rare to have an unbounded integer or string! Generally constrained in some way:
  - Emails must not be empty, must match a pattern
  - CustomerIds must be positive

```
type Something =
    | ChoiceA of A

type Email =
    | Email of string

type CustomerId =
    | CustomerId of int
```

One choice only?
Why?

Is an EmailAddress just a string?

Is a CustomerId just a int?

Use single choice types to keep them distinct

```
type EmailAddress = EmailAddress of string
type PhoneNumber = PhoneNumber of string

type CustomerId = CustomerId of int
type OrderId = OrderId of int
```

Distinct types

Also distinct types

```
let createEmailAddress (s:string) =
    if Regex.IsMatch(s,@"^\S+@\S+\.\S+$")
        then Some (EmailAddress s)
        else None

createEmailAddress:
        string –› EmailAddress option
```

```
type String50 = String50 of string

let createString50 (s:string) =
    if s.Length <= 50
        then Some (String50 s)
        else None

createString50 :
        string –› String50 option
```

What's wrong with this picture?

How could this happen?

How many people ever do this?

New type just for this domain

```
type OrderLineQty = OrderLineQty of int

let createOrderLineQty qty =
    if qty >0 && qty <= 99
        then Some (OrderLineQty qty)
        else None

createOrderLineQty:
        int –› OrderLineQty option
```

Review

ExamplesFromSlides.fsx
OptionTest.fsx
ConstrainedType.fsx

Exercise 02:

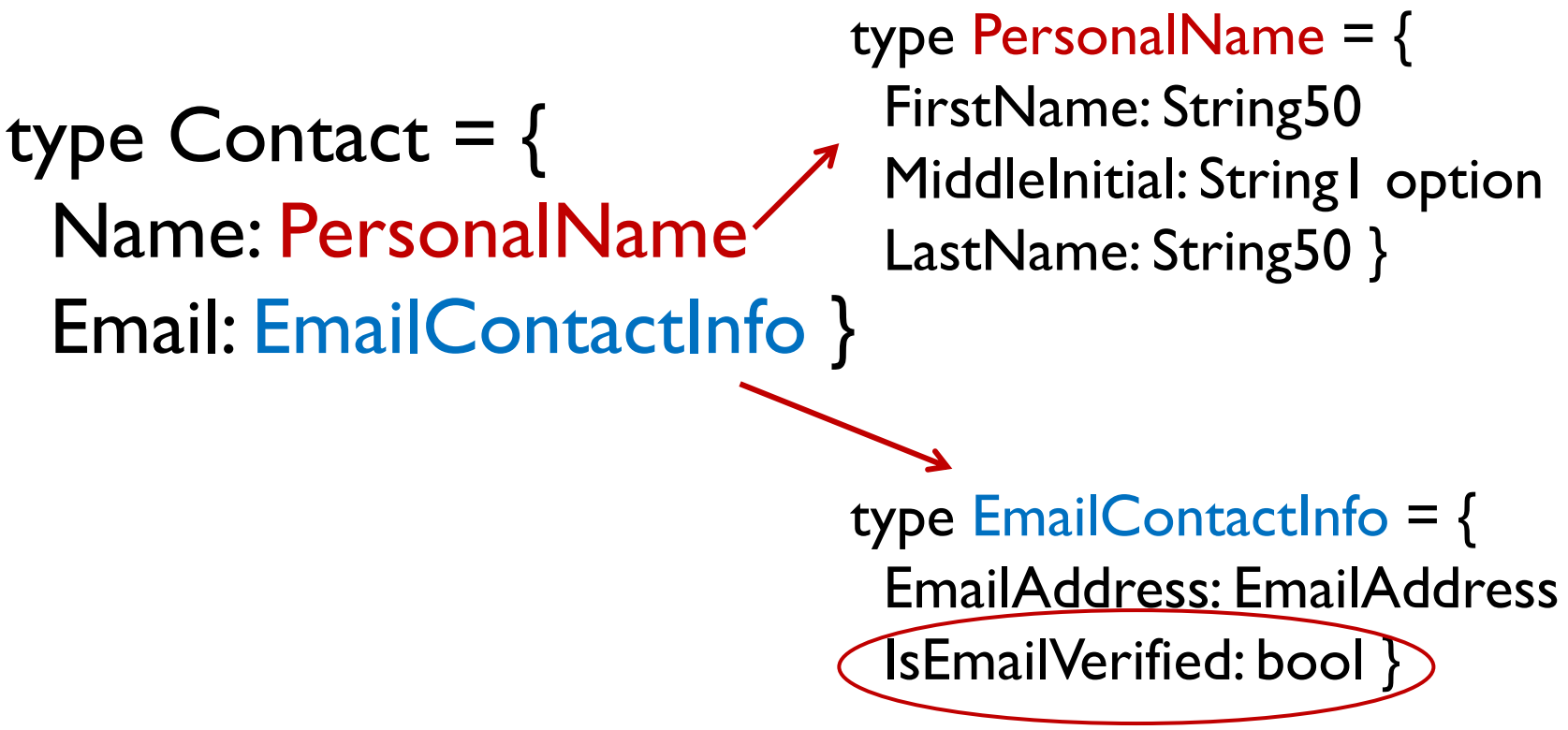
Exercise-OrderLineQty.fsx
Exercise-ConstrainedTypes.fsx

The "Contact" challenge, revisited

```
type Contact = {

  FirstName: string
  MiddleInitial: string
  LastName: string

  EmailAddress: string
  IsEmailVerified: bool
  }
```

```
type Contact = {

  FirstName: string
  MiddleInitial: string option
  LastName: string

  EmailAddress: string
  IsEmailVerified: bool
  }
```

```
type Contact = {

  FirstName: String50
  MiddleInitial: String1  option
  LastName: String50

  EmailAddress: EmailAddress
  IsEmailVerified: bool
  }
```


type Contact = {
Name: PersonalName
Email: EmailContactInfo }
type PersonalName = {
FirstName: String50
MiddleInitial: String1 option
LastName: String50 }
type EmailContactInfo = {
EmailAddress: EmailAddress
IsEmailVerified: bool }

Replacing flags with choices

```
type EmailContactInfo = {
  EmailAddress: EmailAddress
  IsEmailVerified: bool }
```

- *Rule 1: If the email is changed, the verified flag must be reset to false.*
- *Rule 2: The verified flag can only be set by a special verification service*

"there is no problem that can't be solved by wrapping it in another type"

```
type VerifiedEmail = VerifiedEmail of EmailAddress

type VerificationService =
    (EmailAddress * VerificationHash) -> VerifiedEmail option
```

You give me this

And I *might* give you this

```
type VerifiedEmail = VerifiedEmail of EmailAddress

type VerificationService =
  (EmailAddress * VerificationHash) –> VerifiedEmail option

type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmail
```

A choice of one or the other

The "Contact" challenge, completed

```
type EmailAddress = ...

type VerifiedEmail =
  VerifiedEmail of EmailAddress

type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmail
```

```
type PersonalName = {
  FirstName: String50
  MiddleInitial: String1 option
  LastName: String50 }

type Contact = {
  Name: PersonalName
  Email: EmailContactInfo }
```

```
type EmailAddress = ...

type VerifiedEmail =
  VerifiedEmail of EmailAddress

type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmail
```

```
type PersonalName = {
  FirstName: String50
  MiddleInitial: String1 option
  LastName: String50 }

type Contact = {
  Name: PersonalName
  Email: EmailContactInfo }
```

Which values are optional?

What are the constraints?

Which fields are linked?

Domain logic clear?

```
type EmailAddress = ...

type VerifiedEmail =
  VerifiedEmail of EmailAddress

type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmail
```

```
type PersonalName = {
  FirstName: String50
  MiddleInitial: String1 option
  LastName: String50 }

type Contact = {
  Name: PersonalName
  Email: EmailContactInfo }
```

Which values are optional?

```
type EmailAddress = ...

type VerifiedEmail =
  VerifiedEmail of EmailAddress

type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmail
```

```
type PersonalName = {
  FirstName: String50
  MiddleInitial: String1 option
  LastName: String50 }

type Contact = {
  Name: PersonalName
  Email: EmailContactInfo }
```

What are the constraints?

```
type EmailAddress = ...

type VerifiedEmail =
  VerifiedEmail of EmailAddress

type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmail
```

```
type PersonalName = {
  FirstName: String50
  MiddleInitial: String1 option
  LastName: String50 }

type Contact = {
  Name: PersonalName
  Email: EmailContactInfo }
```

Which fields are linked?

```
type EmailAddress = ...

type VerifiedEmail =
  VerifiedEmail of EmailAddress

type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmail
```

```
type PersonalName = {
  FirstName: String50
  MiddleInitial: String1 option
  LastName: String50 }

type Contact = {
  Name: PersonalName
  Email: EmailContactInfo }
```

Domain logic clear?

```
type EmailAddress = ...

type VerifiedEmail =
  VerifiedEmail of EmailAddress

type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmail
```

```
type PersonalName = {
  FirstName: String50
  MiddleInitial: String1 option
  LastName: String50 }

type Contact = {
  Name: PersonalName
  Email: EmailContactInfo }
```

The ubiquitous language is evolving along with the design

(all this is compilable code, BTW)

Review

ExamplesFromSlides.fsx

Exercise 3:

DDD Exercise 1 – CardGame.fsx

DDD Exercise 2 - Contact.fsx

DDD Exercise 3 - Payments.fsx

DDD Exercise 4 - Refactoring flags.fsx

Making illegal states unrepresentable

```
type Contact = {
    Name: Name
    Email: EmailContactInfo
    Address: PostalContactInfo
    }
```

Added some time later

New rule:

"*A contact must have an email or a postal address*"

```
type Contact = {
    Name: Name
    Email: EmailContactInfo
    Address: PostalContactInfo
    }
```

Doesn't meet new requirements

New rule:

"A contact must have an email or a postal address"

```
type Contact = {
    Name: Name
    Email: EmailContactInfo option
    Address: PostalContactInfo option
    }
```

Doesn't meet new requirements either

Could both be missing?

"Make illegal states unrepresentable!"

– Yaron Minsky

"*A contact must have an email or a postal address*"

implies:

- email address only, *or*
- postal address only, *or*
- both email address and postal address

only three possibilities

"*A contact must have an email or a postal address*"

```
type ContactInfo =
    | EmailOnly of EmailContactInfo
    | AddrOnly of PostalContactInfo
    | EmailAndAddr of EmailContactInfo * PostalContactInfo
```

requirements are now encoded in the type!

only three possibilities

```
type Contact = {
    Name: Name
    ContactInfo : ContactInfo  }
```

"A contact must have an email or a postal address"

BEFORE: Email and address separate

```
type Contact = {
   Name: Name
   Email: EmailContactInfo
   Address: PostalContactInfo
}
```

AFTER: Email and address merged into one type

```
type Contact = {
   Name: Name
   ContactInfo : ContactInfo  }

type ContactInfo =
    | EmailOnly of EmailContactInfo
    | AddrOnly of PostalContactInfo
    | EmailAndAddr of
        EmailContactInfo * PostalContactInfo
```

Static types are almost as awesome as this

Is this really what the business wants?

"*A contact must have at least one way of being contacted*"

```
type ContactInfo =
    | Email of EmailContactInfo
    | Addr of PostalContactInfo
```

Way of being contacted

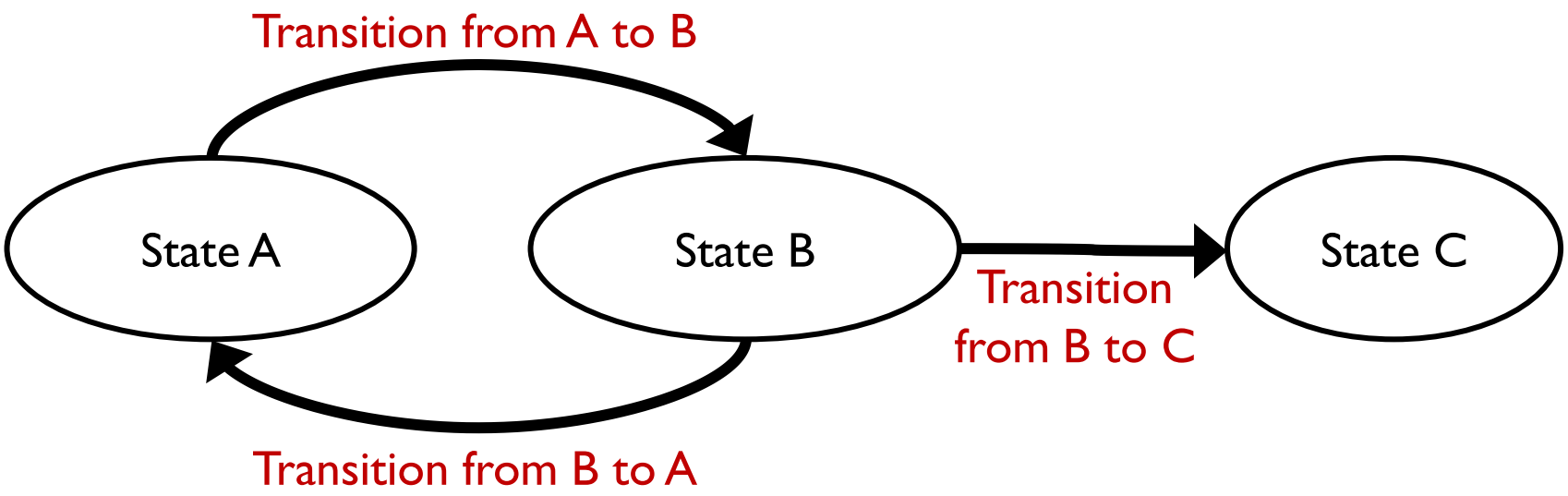
```
type Contact = {
   Name: Name
   PrimaryContactInfo: ContactInfo
   SecondaryContactInfo: ContactInfo option }
```

One way of being contacted is required

Modelling a common scenario

STATES AND TRANSITIONS

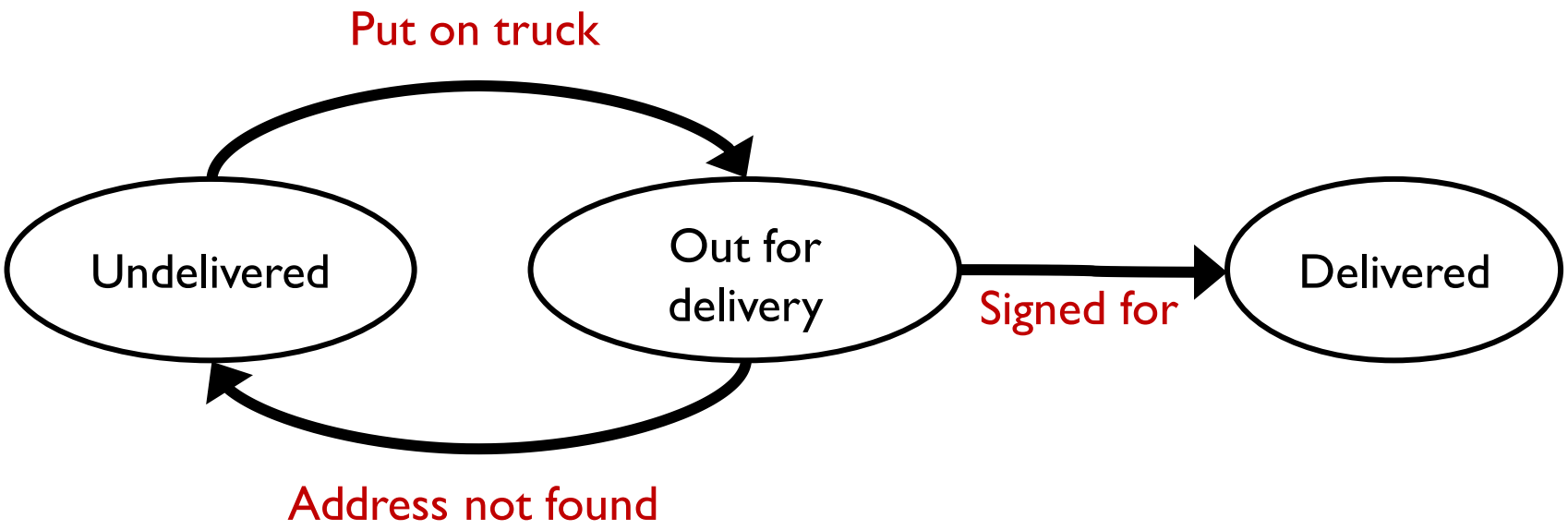
States and transitions



States and transitions for email address

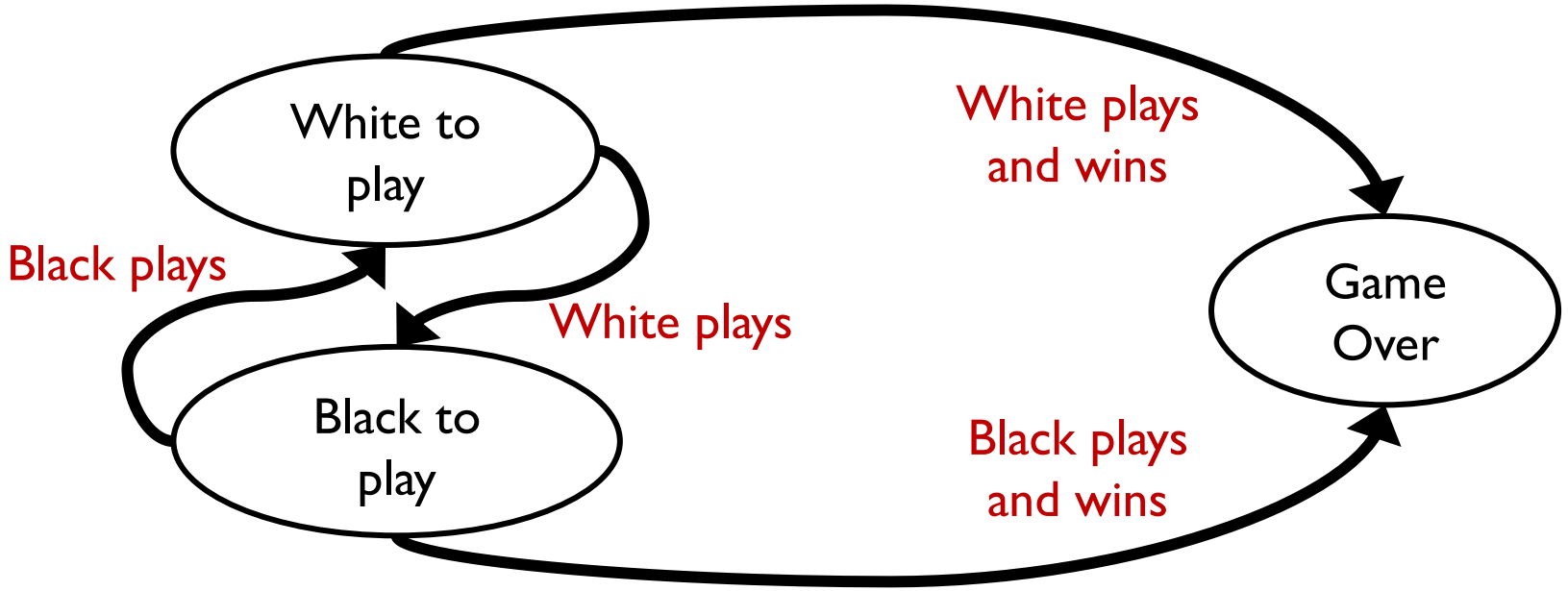
Rule: "You can't send a verification message to a verified email"
Rule: "You can't send a password reset message to a unverified email "

States and transitions for shipments



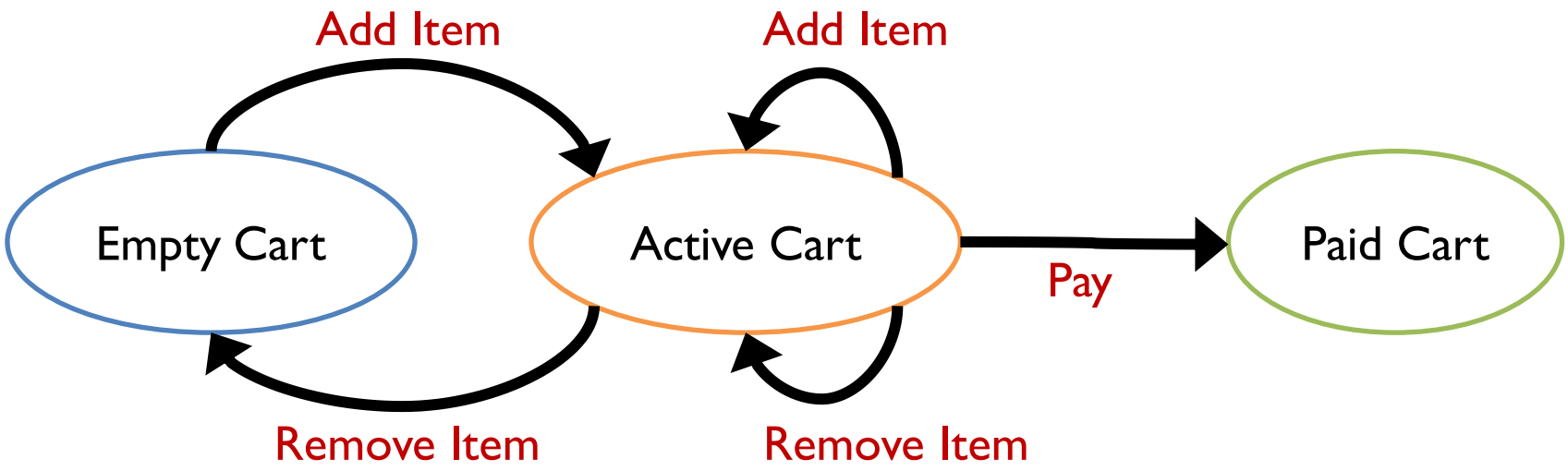
Rule: "You can't put a package on a truck if it is already out for delivery"
Rule: "You can't sign for a package that is already delivered"

States and transitions for chess game



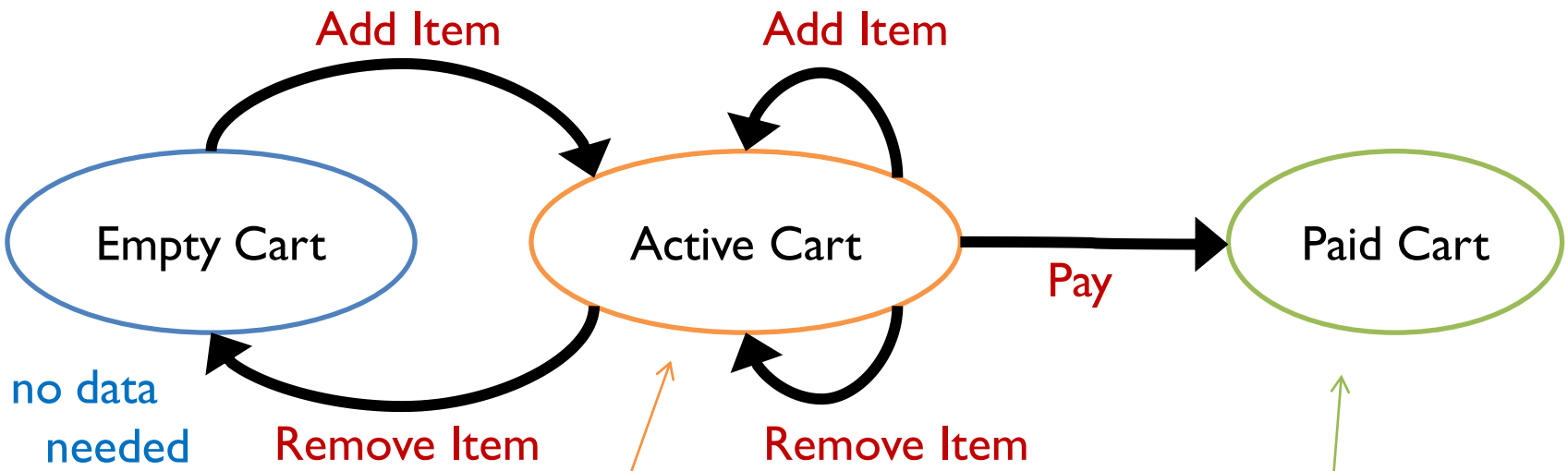
Rule: "White and Black take turns playing.
White can't play if it is Black's turn and vice versa"
Rule: "No one can play when the game is over"

States and transitions for shopping cart



Rule: "You can't remove an item from an empty cart"
Rule: "You can't change a paid cart"
Rule: "You can't pay for a cart twice"

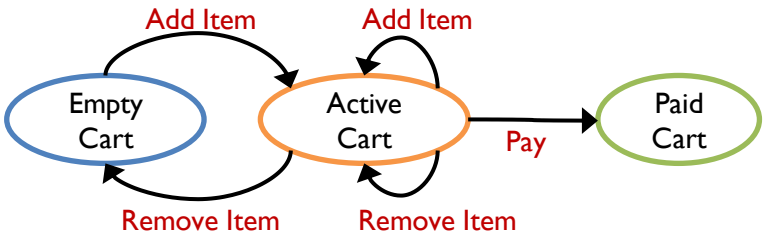
States and transitions for shopping cart




```
type ActiveCartData =
    { UnpaidItems: Item list }
```

```
type PaidCartData =
  { PaidItems: Item list;
    Payment: Payment }
```

What data do we need to store?

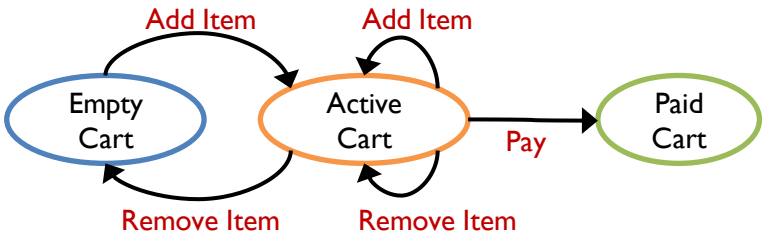



```
type ActiveCartData =
    { UnpaidItems: Item list }
type PaidCartData =
    { PaidItems: Item list; Payment: Payment}

type ShoppingCart =
    | EmptyCart  // no data
    | ActiveCart of ActiveCartData
    | PaidCart of PaidCartData
```

One of three states

No data needed for empty cart state

Shopping Cart API



initCart :

Item –› ShoppingCart

addToActive:

(ActiveCartData * Item) –› ShoppingCart

removeFromActive:

(ActiveCartData * Item) –› ShoppingCart

might be empty or active – can't tell

pay:

(ActiveCartData * Payment) –› ShoppingCart

Server code to add an item

```
let initCart item =
    { UnpaidItems=[item] }
```

create a new ActiveCart with list of one item

```
let addToActive (cart:ActiveCart) item =
    { cart with UnpaidItems = item :: cart.existingItems }
```

Prepends item to list

Client code to add an item using the API

```
let addItem cart item =
    match cart with
    | EmptyCart –›
        initCart item
    | ActiveCart activeData –›
        addToActive(activeData,item)
    | PaidCart paidData –›
        ???
```

Cannot accidentally alter a paid cart!

Server code to remove an item

```
let removeFromActive (cart:ActiveCart) item =
    let remainingItems =
        removeFromList cart.existingItems item
    match remainingItems with
    | [ ] →
        EmptyCart
    | _ →
        {cart with UnpaidItems = remainingItems}
```

create a new ActiveCart with the item removed

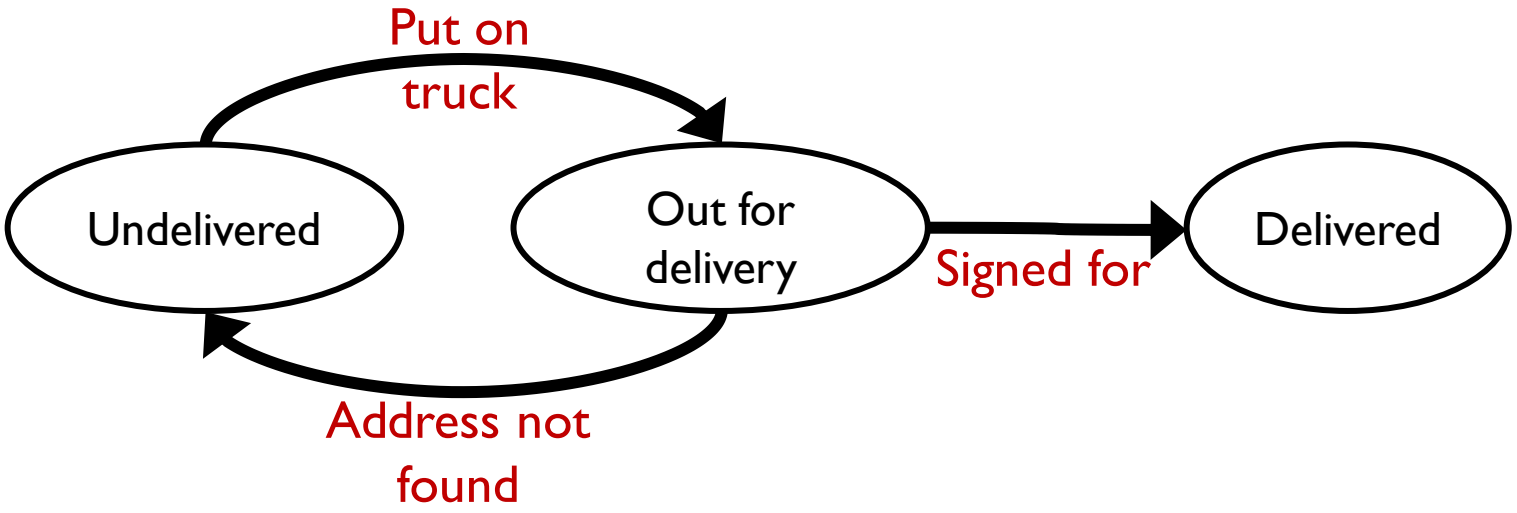
Client code to remove an item using the API

```
let removeItem cart item =
    match cart with
    | EmptyCart –›
        ???
    | ActiveCart activeData –›
        removeFromActive(activeData,item)
    | PaidCart paidData –›
        ???
```

Compiler will not let you remove from an empty cart!

Why design with state transitions?

- Each state can have different allowable data.
- All states are explicitly documented.
- All transitions are explicitly documented.
- It is a design tool that forces you to think about every possibility that could occur.



Review last part of

`ExamplesFromSlides.fsx`

Exercise 4:

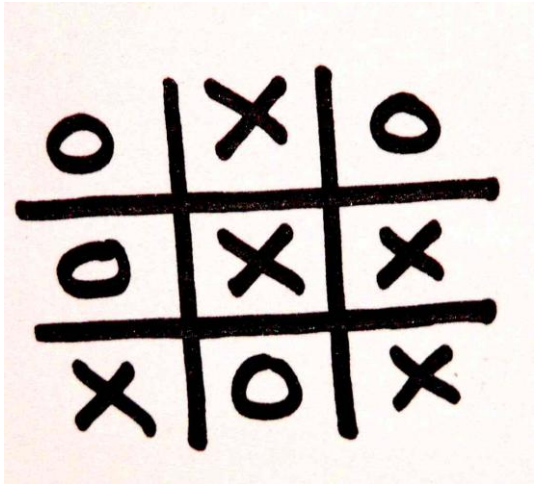
`FSM Exercise 1 - Verified Email.fsx`

`FSM Exercise 2 - Shopping cart.fsx DDD`

`FSM Exercise 4 - Shipments.fsx`

Exercise 5:
Convert the domain models to types

Convert the domain models to types



Summary: The power of modeling with types

A real world example from the DDD book

A real world example from the DDD book

```
module Cargo =
  type TrackingId = TrackingId  of string
  type Location = Location of string

  type RouteSpecification = {
     Origin: Location
     Destination: Location }

  type TransportStatus =
     Claimed | NotReceived
     | InPort | OnboardCarrier | Unknown

  type Cargo = {
     RouteSpecification : RouteSpecification
     }

  type TrackedCargo = {
    TrackingId : TrackingId
    Cargo : Cargo }
```

A real world example from the DDD book

```
module Cargo =
  type TrackingId = TrackingId  of string
  type Location = Location of string

  type RouteSpecification = {
     Origin: Location
     Destination: Location }

  type TransportStatus =
     Claimed | NotReceived
     | InPort | OnboardCarrier | Unknown

  type Cargo = {
     RouteSpecification : RouteSpecification
     }

  type TrackedCargo = {
    TrackingId :TrackingId
    Cargo : Cargo }
```

```
type Leg = {
    LoadLocation : Location
    UnloadLocation : Location
    LoadTime : DateTime
    UnloadTime : DateTime }

type Itinerary = Leg list

type RoutedCargo = {
  Itinerary : Itinerary
  Cargo :TrackedCargo }

type Track =
   Cargo * TrackingId   –› TrackedCargo

type Route =
  TrackedCargo * Policy –› RoutedCargo
```

A real world example from the DDD book

```
module Cargo =
  type TrackingId = TrackingId  of string
  type Location = Location of string

  type RouteSpecification = {
     Origin: Location
     Destination: Location }

  type TransportStatus =
     Claimed | NotReceived
     | InPort | OnboardCarrier | Unknown

  type Cargo = {
     RouteSpecification : RouteSpecification
     }

  type TrackedCargo = {
    TrackingId :TrackingId
    Cargo : Cargo }
```

Nouns

```
type Leg = {
    LoadLocation : Location
    UnloadLocation : Location
    LoadTime : DateTime
    UnloadTime : DateTime }

type Itinerary = Leg list

type RoutedCargo = {
  Itinerary : Itinerary
  Cargo :TrackedCargo }
```

Verbs

```
type Track =
   Cargo * TrackingId   –› TrackedCargo

type Route =
   TrackedCargo * Policy –› RoutedCargo
```

This is just a start – it could be much more detailed!

"Value working software over comprehensive documentation"

So how best to document our designs?

"Value working software over comprehensive documentation."

Good documentation should be:

- Trustworthy
- Easy to change
- Accessible

http://thepaulrayner.com/blog/2013/05/07/succeeding-with-ddd-documentation/

"Value working software over comprehensive documentation."

Good documentation should be:

- Trustworthy
- Easy to change
- Accessible

If the design IS the code, then it can never be out of date.

http://thepaulrayner.com/blog/2013/05/07/succeeding-with-ddd-documentation/

"Value working software over comprehensive documentation."

Good documentation should be:

- Trustworthy
- Easy to change
- Accessible

All domain definitions stored in one file.

http://thepaulrayner.com/blog/2013/05/07/succeeding-with-ddd-documentation/

"Value working software over comprehensive documentation."

Good documentation should be:

- Trustworthy
- Easy to change
- Accessible

It's stored with the code, versioned in github, etc.

http://thepaulrayner.com/blog/2013/05/07/succeeding-with-ddd-documentation/

Reason 2.
Types encourage
accurate domain modelling

Business rule:

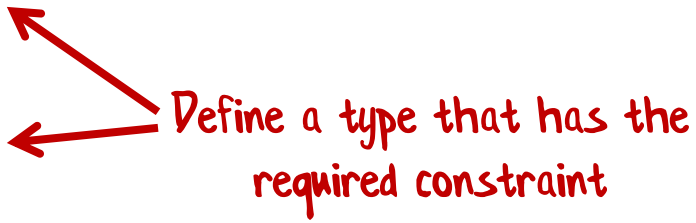
"First and last name must not be more than 50 chars"

```
type Contact = {
                Must not be more than 50 chars
  FirstName: string
  MiddleInitial: string
  LastName: string

  EmailAddress: string
  IsEmailVerified: bool
  }
```

Business rule:

"First and last name must not be more than 50 chars"

```
type Contact = {

  FirstName: String50
  MiddleInitial: String1
  LastName: String50

  EmailAddress: string
  IsEmailVerified: bool
  }
```



Business rule:

"Email field must be a valid email address"

```
type Contact = {

  FirstName: String50
  MiddleInitial: String1
  LastName: String50

  EmailAddress: string
  IsEmailVerified: bool
  }
```

Must contain an "@" sign

Business rule:

"Email field must be a valid email address"

```
type Contact = {

  FirstName: String50
  MiddleInitial: String1
  LastName: String50

  EmailAddress: EmailAddress
  IsEmailVerified: bool
  }
```

Define a type that has the required constraint

Business rule:

"Middle initial is optional"

```
type Contact = {

  FirstName: String50
  MiddleInitial: String1
  LastName: String50

  EmailAddress: EmailAddress
  IsEmailVerified: bool
  }
```

Required?

Business rule:

"Middle initial is optional"

```
type Contact = {

  FirstName: String50
  MiddleInitial: String1 option
  LastName: String50

  EmailAddress: EmailAddress
  IsEmailVerified: bool
  }
```

Business rule:

"Verified emails are different from unverified emails"

```
type Contact = {

  FirstName: String50
  MiddleInitial: String1  option
  LastName: String50

  EmailAddress: EmailAddress
  IsEmailVerified: bool
  }
```

What is the business logic?

Business rule:

"Verified emails are different from unverified emails"

```
type EmailAddress = ...

type VerifiedEmail =
  VerifiedEmail of EmailAddress

type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmail
```

Represent with choice type

Business rule:

"Verified emails are different from unverified emails"

```
type EmailAddress = ...

type VerifiedEmail =
  VerifiedEmail of EmailAddress

type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmail
```

Better modelling

```
type Contact = {

  FirstName: String50
  MiddleInitial: String1 option
  LastName: String50

  EmailAddress: EmailContactInfo
  }
```

And boolean has gone!

But wait! There's more!

Reason 3.
Types can encode business rules

"compile time unit tests"

Business rule:

"A contact must have an email or a postal address"

```
type Contact = {
   Name: Name
   Email: EmailContactInfo
   Address: PostalContactInfo
   }
```

Doesn't meet new requirements...

because the design implies both are required.

Business rule:

"A contact must have an email or a postal address"

```
type Contact = {
   Name: Name
   Email: EmailContactInfo option
   Address: PostalContactInfo option
   }
```

Still doesn't meet new requirements!

Why? Because both could be missing.

"Make illegal states unrepresentable!"

– Yaron Minsky

"A contact must have an email or a postal address"

implies:

- email address only, *or*
- postal address only, *or*
- both email address and postal address

only three possibilities

"A contact must have an email or a postal address"

```
type ContactInfo =
  | EmailOnly of EmailContactInfo
  | AddrOnly of PostalContactInfo
  | EmailAndAddr of EmailContactInfo * PostalContactInfo

type Contact = {
  Name: Name
  ContactInfo : ContactInfo  }
```

only three possibilities

requirements are now encoded in the type!

Static types are almost as awesome as this

Summary: What types are good for

- Types as executable documentation
  - Ubiquitous language
  - Design and code are synchronized
  - Code is understandable by domain expert
- Types for accurate domain modelling
  - Constraints are explicit
- Types can encode business rules
  - Illegal states can be made unrepresentable

Paulmichael Blasucci
@pblasucci

Following

"The domain model [code] is so succinct the business analysts have started using it as documentation."

Simon Cousins
@simontcousins

Following

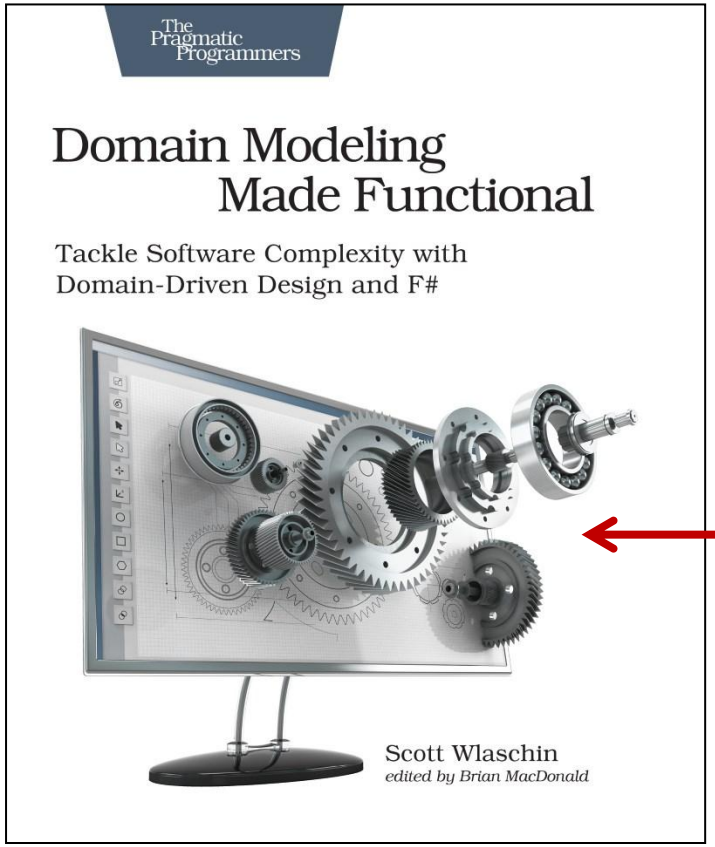
pm: "that code is clearer than the spec", me: "can i paste it into the documentation then?", pm: "yes"

Reid Evans
@ReidNEvans

Follow

We had been having difficulty understanding the domain. 40 lines of #fsharp later we were all on the same page

More on DDD and designing with types at fsharpforfunandprofit.com/ddd



My book all about this!

Review

ExamplesFromSlides.fsx

Exercises:

DDD Exercise 1 – CardGame.fsx

DDD Exercise 2 - Contact.fsx

DDD Exercise 3 - Payments.fsx

DDD Exercise 4 - Refactoring flags.fsx

FSM Exercise 1 - Verified Email.fsx

FSM Exercise 2 - Shopping cart.fsx

FSM Exercise 4 - Shipments.fsx