# 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
# <u>accurate</u> 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
  }
```

Define a type that has the required constraint

Business rule:

*"Email field must be a valid email address"*

type **Contact** = {

FirstName: String50
MiddleInitial: String1
LastName: String50

EmailAddress: string Must contain an "@" sign
IsEmailVerified: bool
}

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 Required?
LastName: String50

EmailAddress: EmailAddress
IsEmailVerified: bool
}

Business rule:

*"Middle initial is optional"*

```
type Contact = {

  FirstName: String50
  MiddleInitial: String1 option
  LastName: String50

  EmailAddress: EmailAddress
  IsEmailVerified: bool
  }
```

Optional can be applied to any type

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
```

only three possibilities

requirements are now encoded in the type!

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

# 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

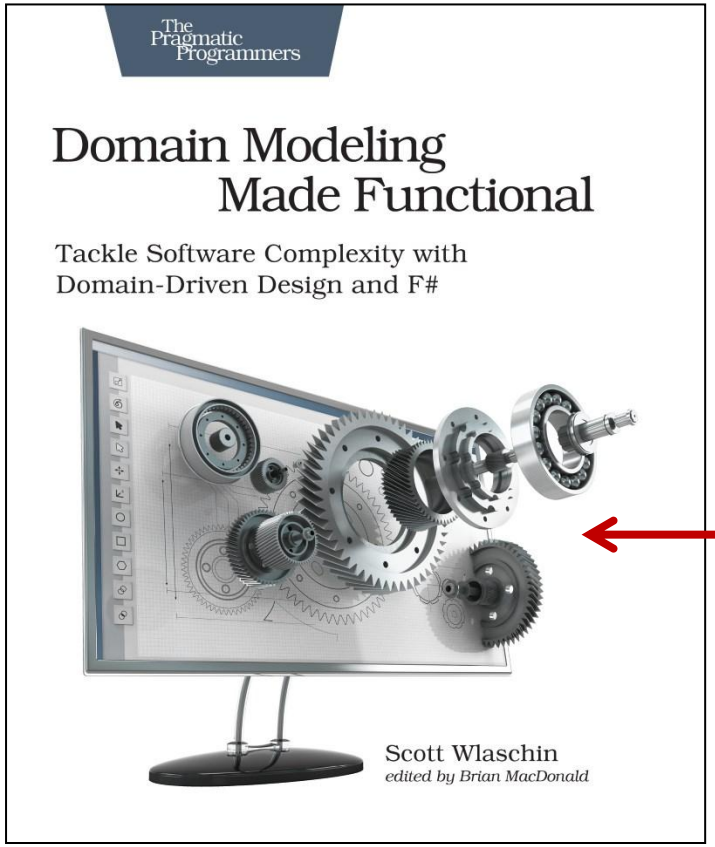
pm: "that code is clearer than the spec", me: "can i paste it into the documentation then?", pm: "yes"

**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!