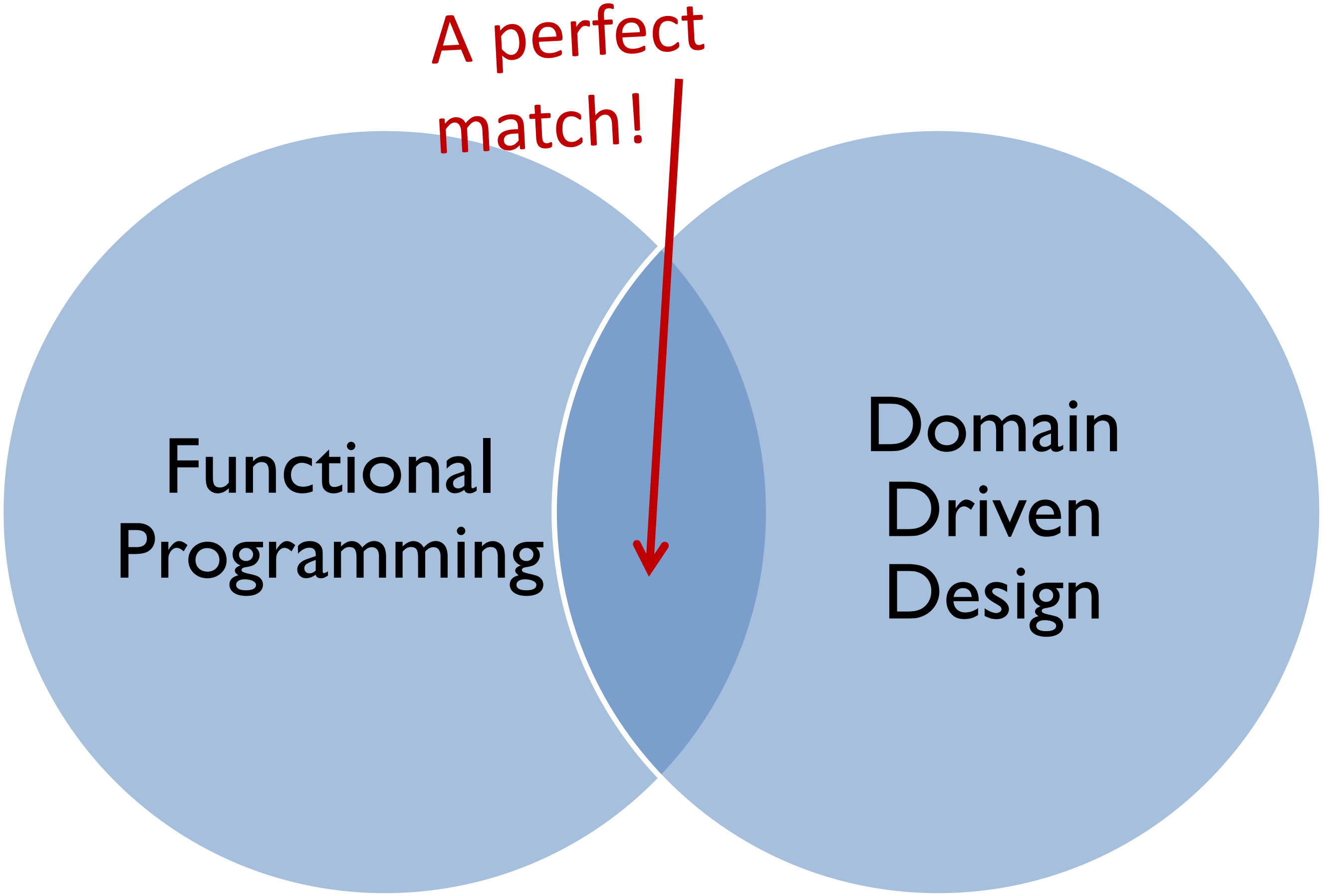
Domain Modeling Made Functional

Functional programming: what is it good for?

- Mathematical things only

Functional programming: what is it good for?

- ~~Mathematical things only~~
- Interactive & collaborative domain modeling
- Representing a domain model accurately


A perfect match!
Functional Programming
Domain Driven Design

Part I

Communication & Feedback

This isn't about coding,
so why should you care?

What's the problem?

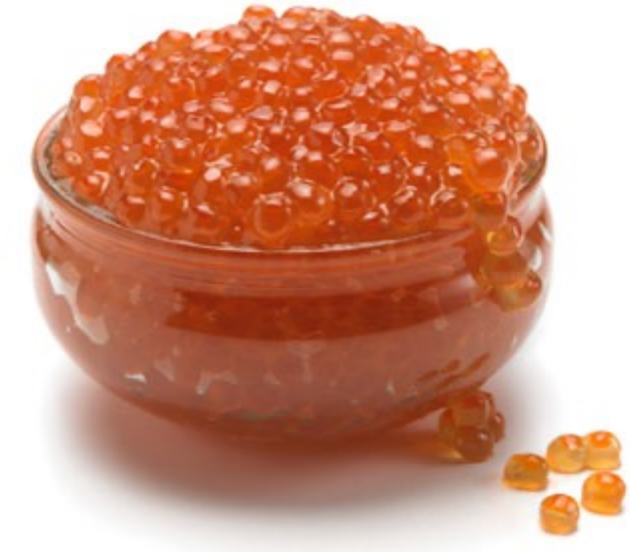
1. Misunderstanding the requirements
2. Acting on the requirements without getting feedback first

Most romantic comedies are based on the same premise.

Pro Tip: we don't want real life to be funny like this.

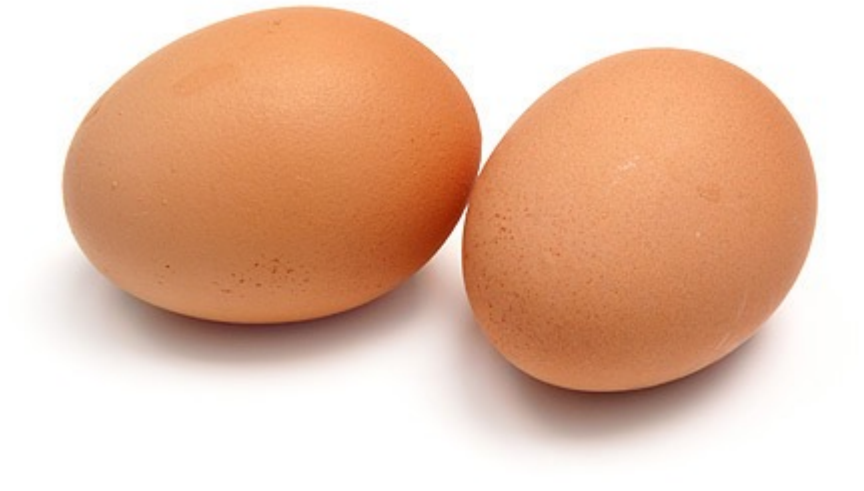
A café example

- Customer: "Can I have some eggs?"
- Waiter to chef: "Some eggs, please"
- Russian chef: "Here you go…"



A café example

- Waiter to chef: "Not fish eggs, chicken eggs "
- Chef: "Ok, here you go…"



A café example

- Waiter to chef: " No, *cooked* chicken eggs"
- Chef: "Ok, this time I understand…"

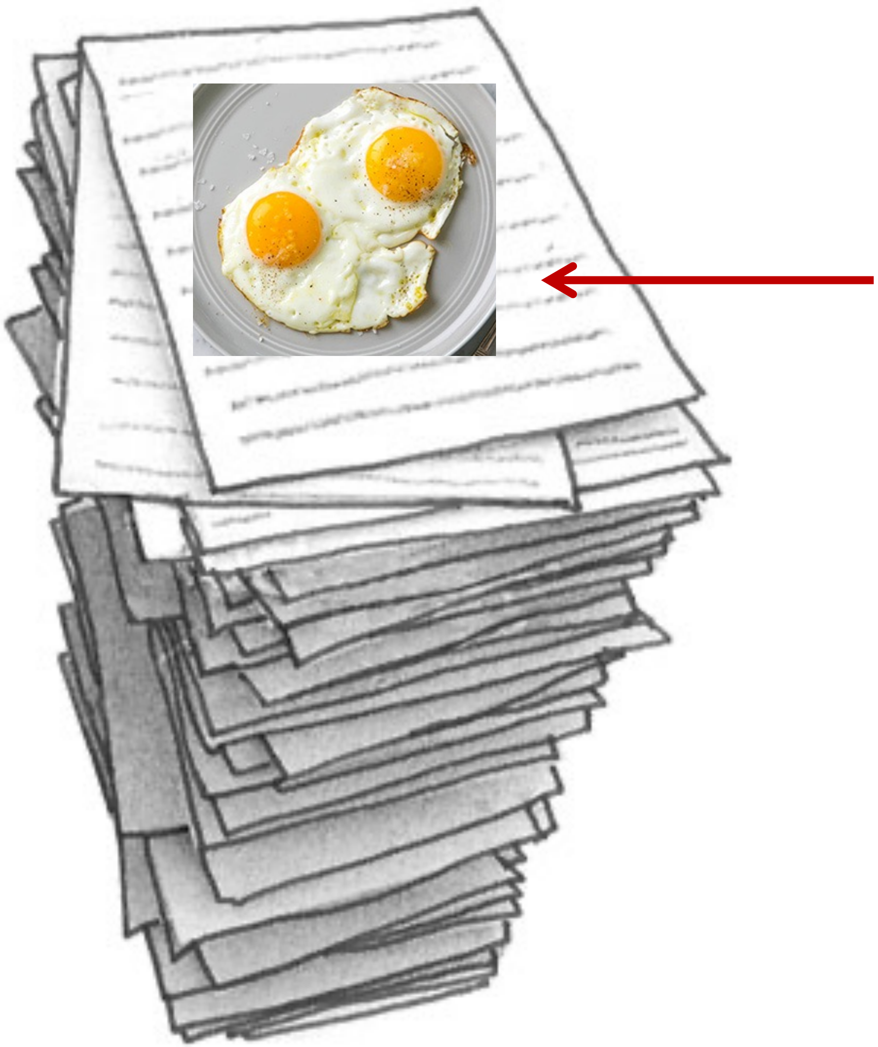
A café example

- Waiter to customer: "Here are your eggs"
- Customer: "I wanted fried eggs"

😢

Miscommunication

What's the solution?



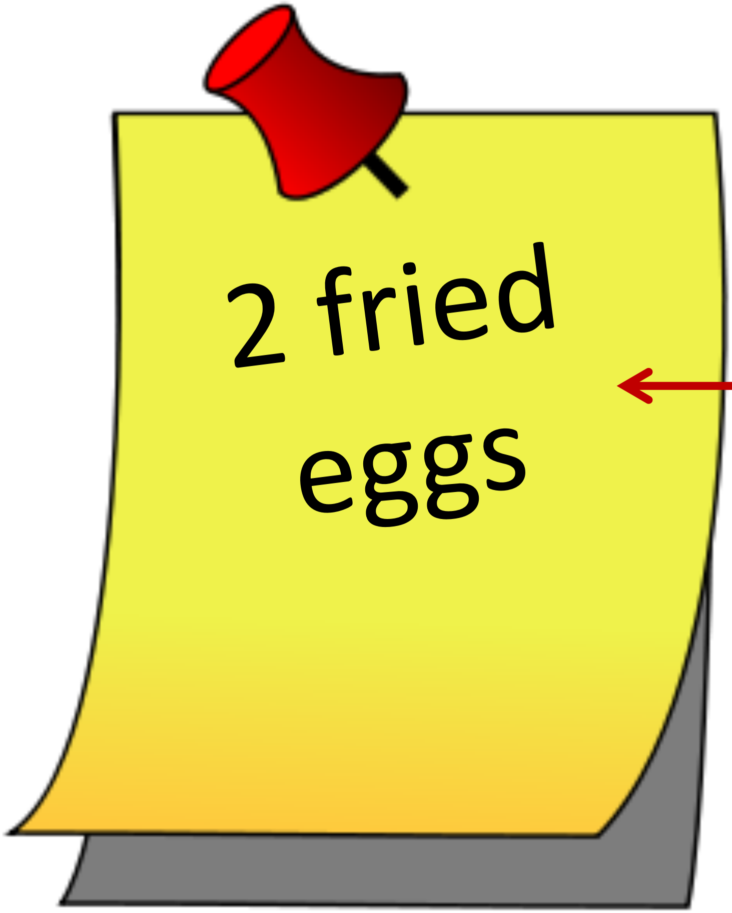
A 200 page spec on how to cook a fried egg

Who thinks this will work?

This is not the solution!

- We expect the chef to be a subject matter expert – an expert on making breakfast
- A 200 page spec should not be needed!

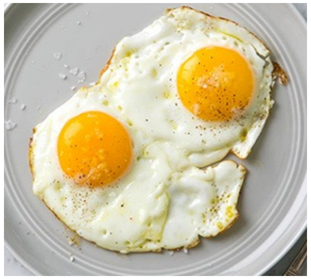
What happens in real life?

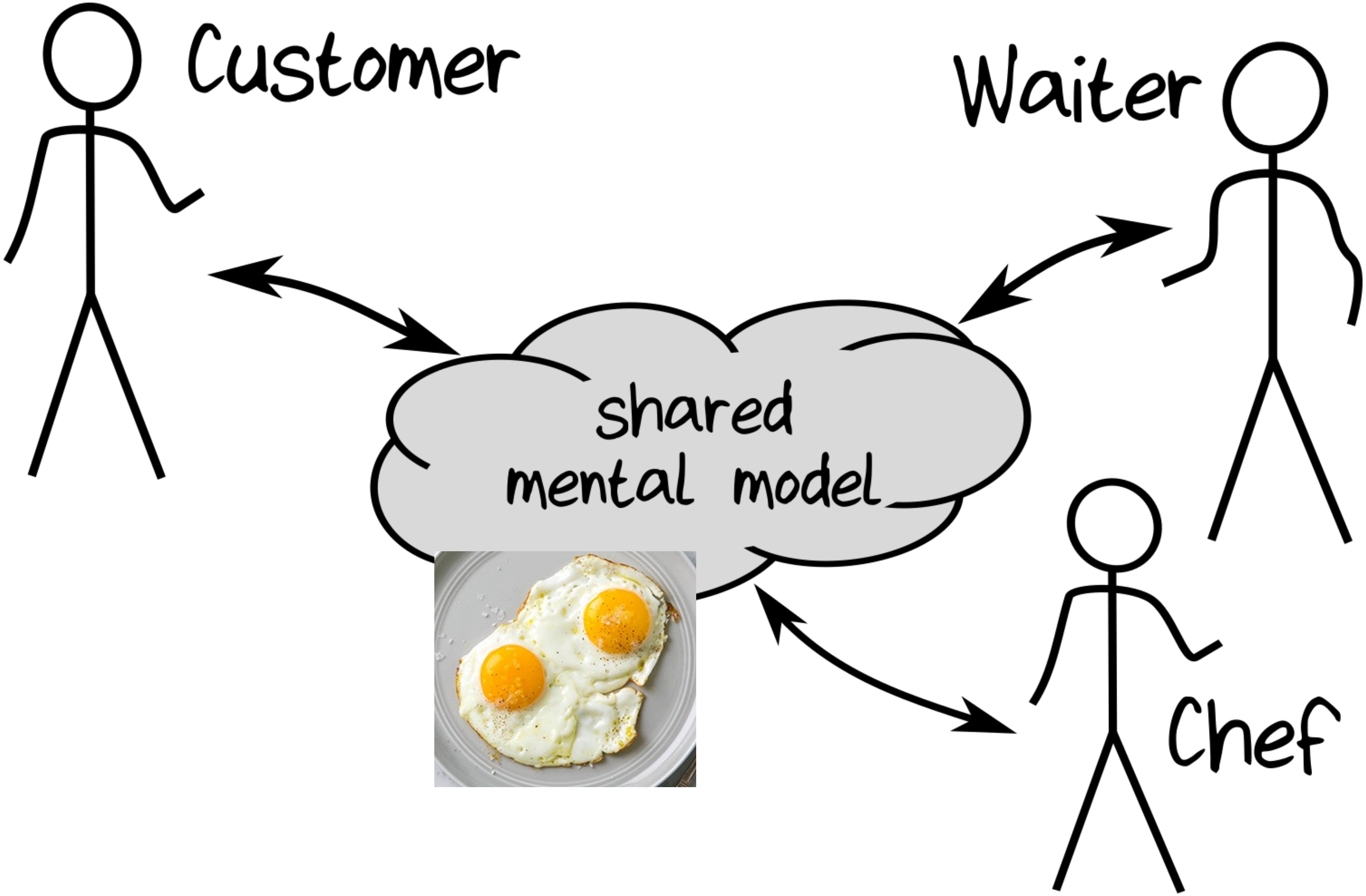


A 3 word spec

Why does this tiny spec work?

- **Shared knowledge** of the domain
  - – *Everyone* is a "breakfast" expert!
- **Shared vocabulary**
  - – Everyone knows what "fried eggs" means.

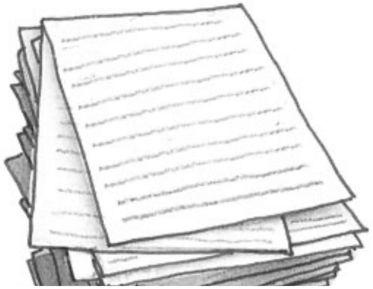



Customer
Waiter
shared
mental model
Chef

How long must a spec be?

- Who here has a specialized hobby/interest?
- If I asked you (an expert) to write an app for me, how big a spec would I need? Why?

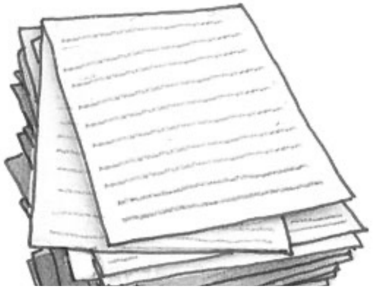
or?



How long must a spec be?

- If I asked a non-expert to write the same app for me, how big a spec would I need? Why?

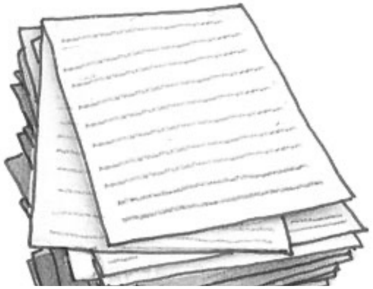
or?



How long must a spec be?

- Which of these two projects is more likely to succeed?
  - Written by the expert or non-expert?

or?



Why are experts better?

- An expert will build the app better
  - And faster
  - With a smaller spec
  - And less confusion

Because…

- **Shared knowledge** of the domain
- **Shared vocabulary**

Part II

Domain Driven Design

What does all this have to do with software projects?

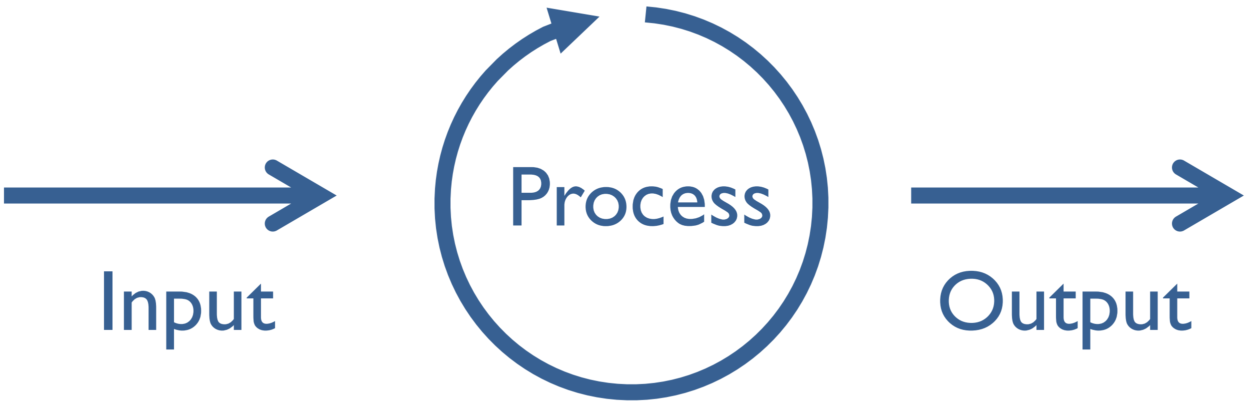
In my experience, most projects fail because:

- Misunderstanding requirements, or
- Going in the wrong direction, or
- Starting off in the right direction but veering off course

What's the ideal software development process?

- Build a shared mental model
  - Become an "expert"
  - Means a smaller spec
  - Less misunderstanding
- *And* have frequent feedback
  - Make sure you are going in the right direction
    - No point going fast in the wrong direction!
  - Do a course correction if goals change

The software development process

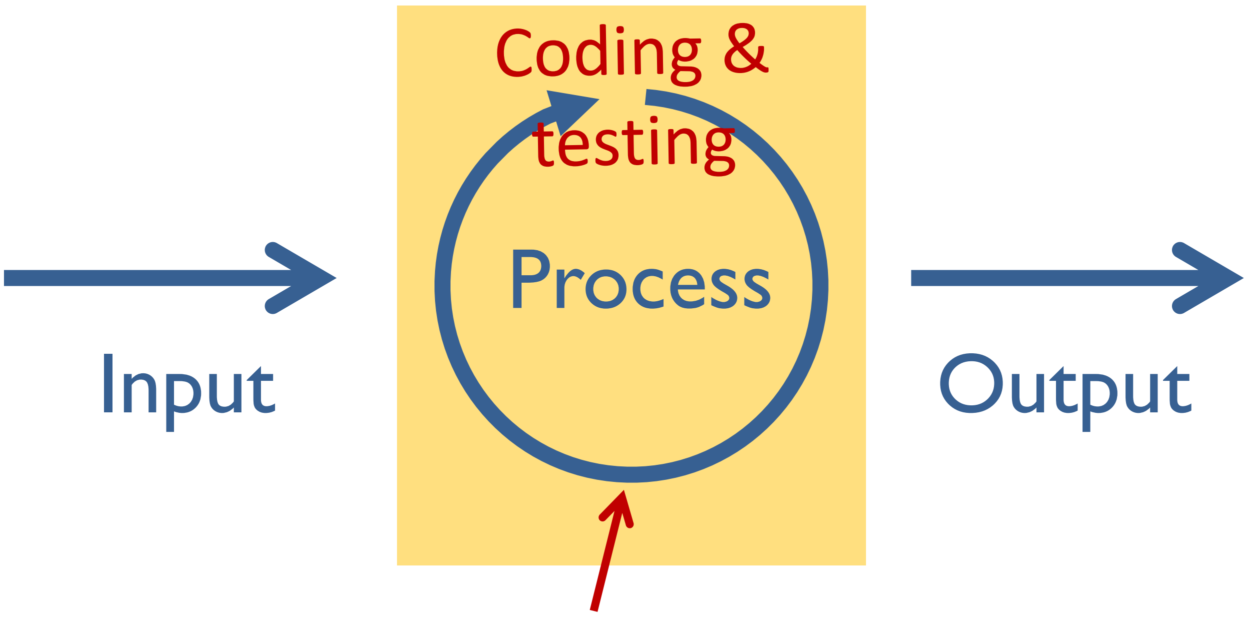


Input

Process

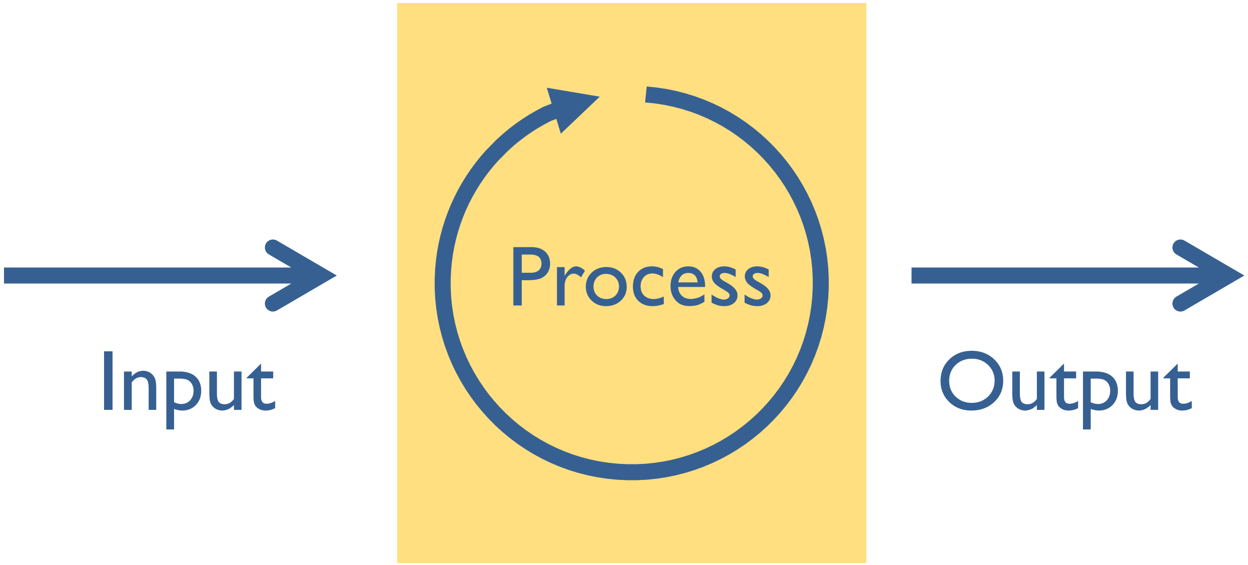
Output

The software development process



Developers love to focus on this

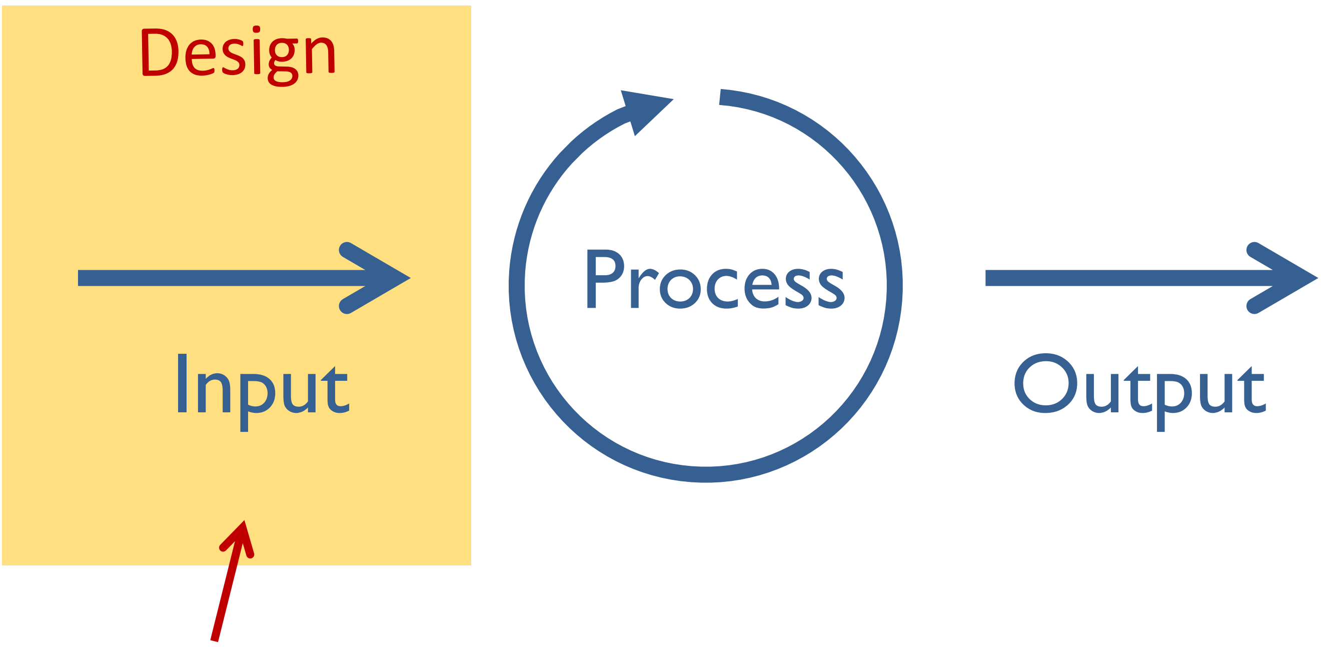
The software development process



Garbage in

Garbage out

The software development process



Design

Input

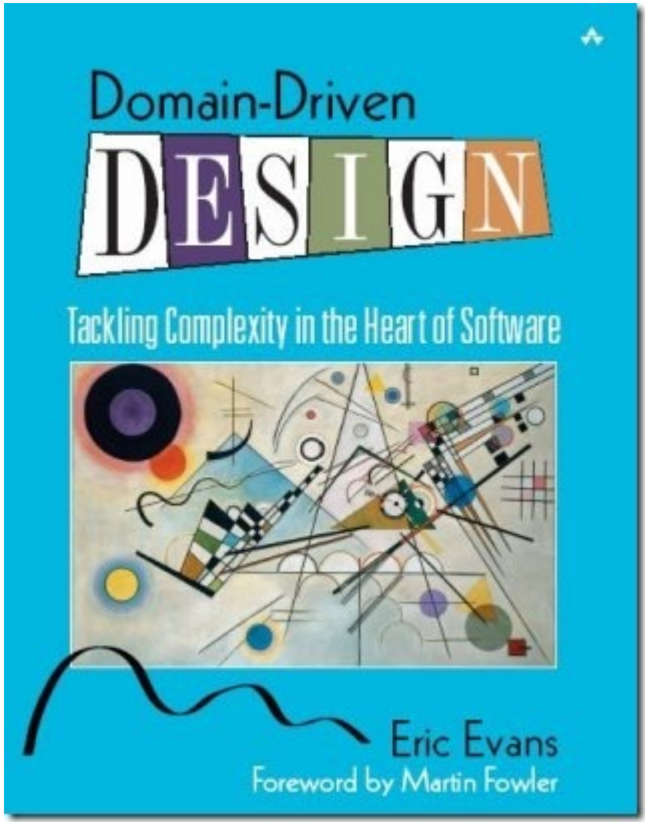
Process

Output

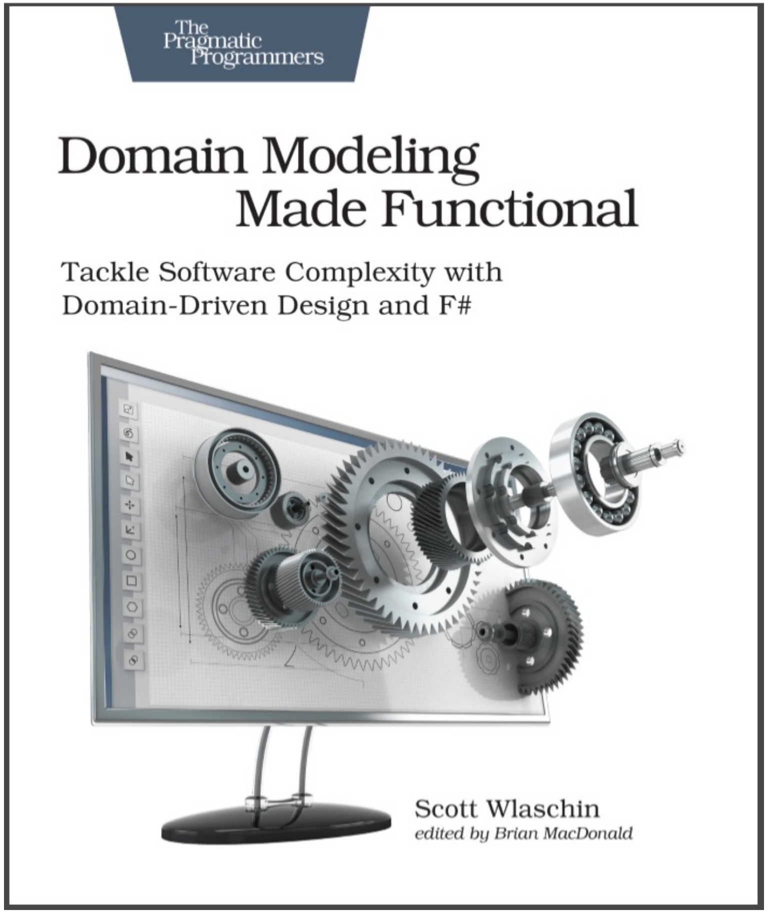
This talk

(reduce) Garbage in (reduced)

Garbage out


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler

"Focus on the domain and domain logic rather than technology"
-- Eric Evans



The Pragmatic Programmers

Domain Modeling Made Functional

Tackle Software Complexity with Domain-Driven Design and F#

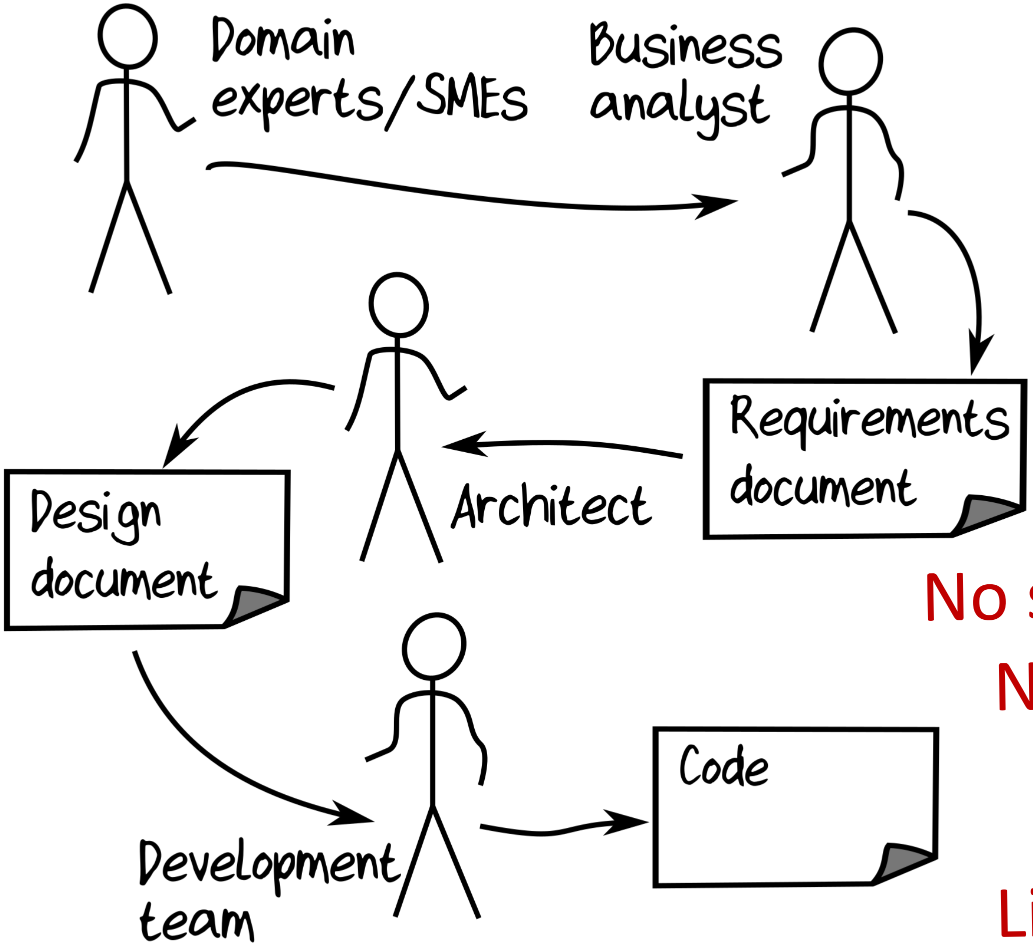
Scott Wlaschin

edited by Brian MacDonald

Or read the first 2 chapters of this book!

Why Domain-Driven Design?

Waterfall

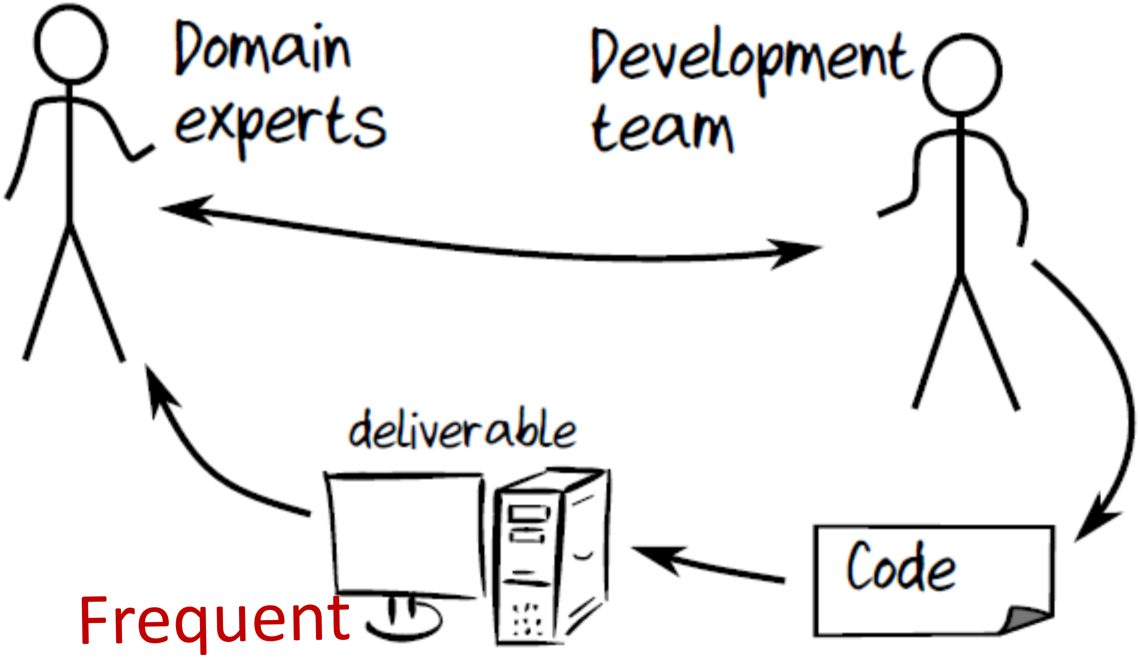


No shared model

No feedback

Like a bad children's game...

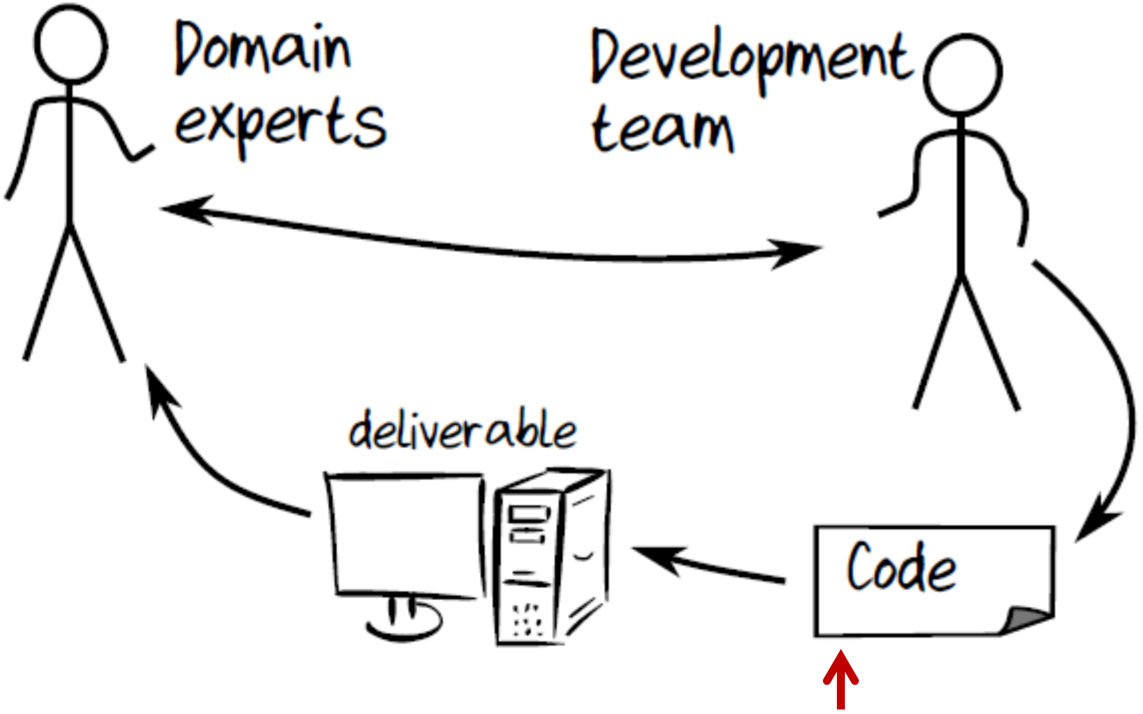
Agile



Frequent deliverables

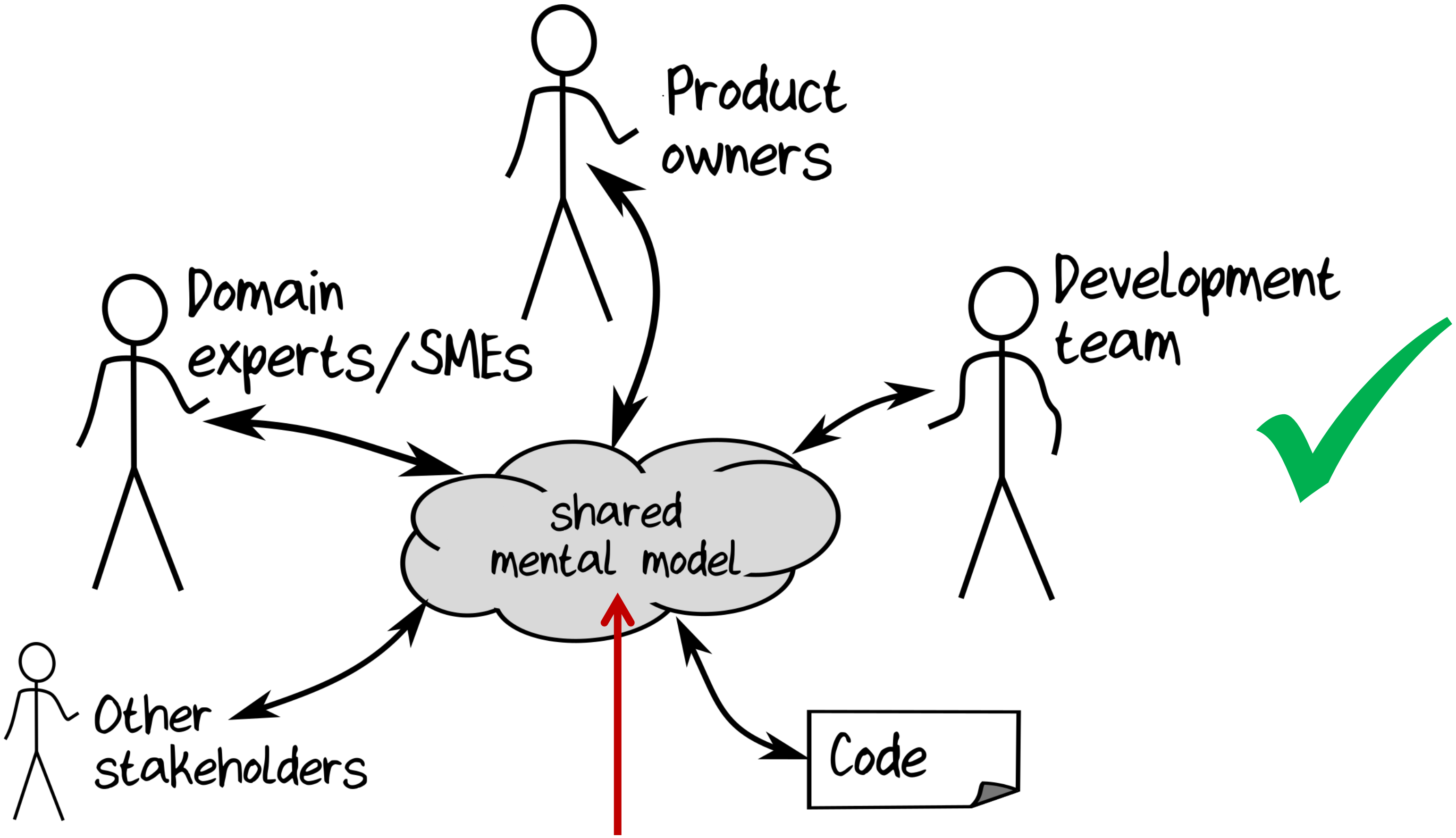
Focus on feedback is good...

Agile

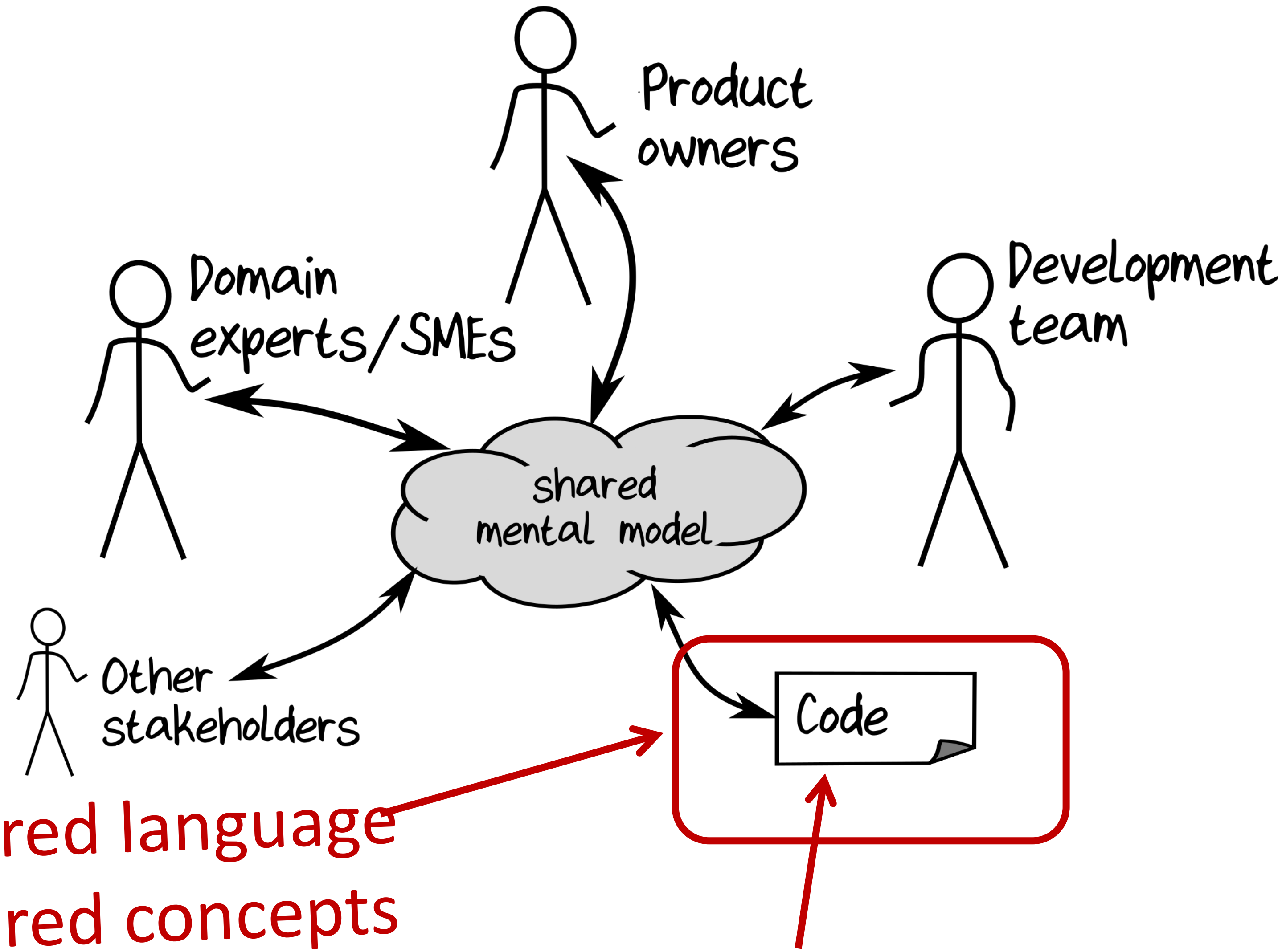


But code is still a "translation"

Domain-Driven Design



shared language
shared concepts



shared language
shared concepts

Domain model == code == documentation

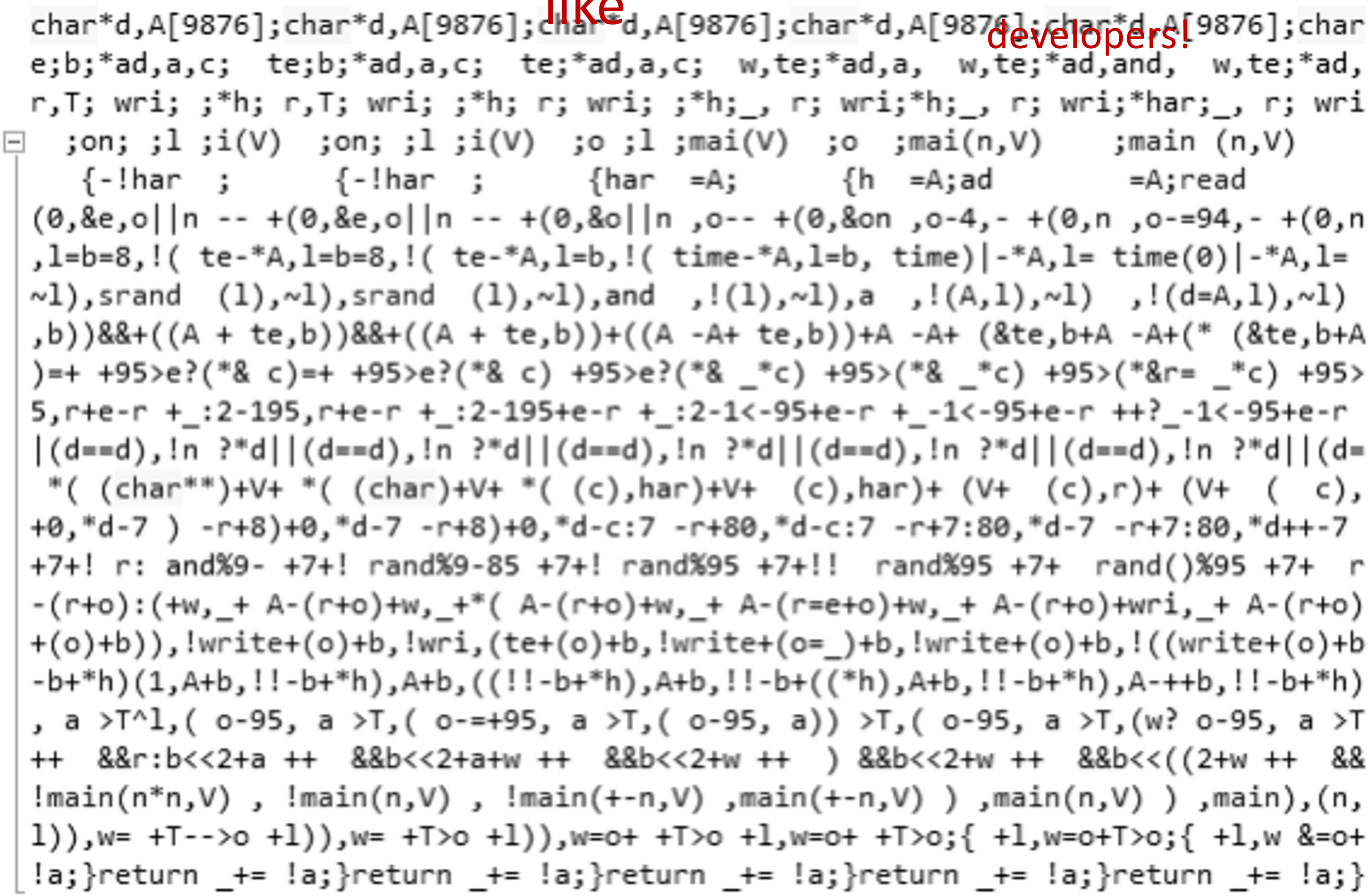
How can we do design right?

- Agile contribution:
  - **Rapid feedback** during design
- DDD contribution:
  - Stakeholders have a **shared mental model**
  - …which is also represented in the **code**

Can you really make code represent the domain?

What non-developers think source code looks like

and some C developers!



```
char*d,A[9876];char*d,A[9876];char*d,A[9876];char*d,A[9876];char*d,A[9876];char
e;b;*ad,a,c;  te;b;*ad,a,c;  te;*ad,a,c;  w,te;*ad,a,  w,te;*ad,and,  w,te;*ad,
r,T; wri; ;*h; r,T; wri; ;*h; r; wri; ;*h;_, r; wri;*h;_, r; wri;*har;_, r; wri
  ;on; ;l ;i(V)  ;on; ;l ;i(V)  ;o ;l ;mai(V)  ;o  ;mai(n,V)    ;main (n,V)
   {-!har  ;      {-!har  ;      {har  =A;      {h  =A;ad        =A;read
(0,&e,o||n -- +(0,&e,o||n -- +(0,&o||n ,o-- +(0,&on ,o-4,- +(0,n ,o-=94,- +(0,n
,l=b=8,!( te-*A,l=b=8,!( te-*A,l=b,!( time-*A,l=b, time)|-*A,l= time(0)|-*A,l=
~l),srand  (l),~l),srand  (l),~l),and  ,!(l),~l),a  ,!(A,l),~l)  ,!(d=A,l),~l)
,b))&&+((A + te,b))&&+((A + te,b))+((A -A+ te,b))+A -A+ (&te,b+A -A+(* (&te,b+A
)=+ +95>e?(*& c)=+ +95>e?(*& c) +95>e?(*& _*c) +95>(*& _*c) +95>(*&r= _*c) +95>
5,r+e-r +_:2-195,r+e-r +_:2-195+e-r +_:2-1<-95+e-r +_-1<-95+e-r ++?_-1<-95+e-r
|(d==d),!n ?*d||(d==d),!n ?*d||(d==d),!n ?*d||(d==d),!n ?*d||(d==d),!n ?*d||(d=
 *( (char**)+V+ *( (char)+V+ *( (c),har)+V+  (c),har)+ (V+  (c),r)+ (V+  (  c),
+0,*d-7 ) -r+8)+0,*d-7 -r+8)+0,*d-c:7 -r+80,*d-c:7 -r+7:80,*d-7 -r+7:80,*d++-7
+7+! r: and%9- +7+! rand%9-85 +7+! rand%95 +7+!!  rand%95 +7+  rand()%95 +7+  r
-(r+o):(+w,_+ A-(r+o)+w,_+*( A-(r+o)+w,_+ A-(r=e+o)+w,_+ A-(r+o)+wri,_+ A-(r+o)
+(o)+b)),!write+(o)+b,!wri,(te+(o)+b,!write+(o=_)+b,!write+(o)+b,!((write+(o)+b
-b+*h)(1,A+b,!!-b+*h),A+b,((!!-b+*h),A+b,!!-b+((*h),A+b,!!-b+*h),A-++b,!!-b+*h)
, a >T^l,( o-95, a >T,( o-=+95, a >T,( o-95, a)) >T,( o-95, a >T,(w? o-95, a >T
++  &&r:b<<2+a ++  &&b<<2+a+w ++  &&b<<2+w ++  ) &&b<<2+w ++  &&b<<((2+w ++  &&
!main(n*n,V) , !main(n,V) , !main(+-n,V) ,main(+-n,V) ) ,main(n,V) ) ,main),(n,
l)),w= +T-->o +l)),w= +T>o +l)),w=o+ +T>o +l,w=o+ +T>o;{ +l,w=o+T>o;{ +l,w &=o+
!a;}return _+= !a;}return _+= !a;}return _+= !a;}return _+= !a;}return _+= !a;}
```

What DDD source code should look like

```
module CardGame =

  type Suit = Club | Diamond | Spade | Heart

  type Rank = Two | Three | Four | Five | Six | Seven | Eight
              | Nine | Ten | Jack | Queen | King

  type Card = Suit * Rank

  type Hand = Card list
  type Deck = Card list

  type Player = {Name:string; Hand:Hand}
  type Game = {Deck:Deck; Players: Player list}

  type Deal = Deck -> (Deck * Card)

  type PickupCard = (Hand * Card) -> Hand
```

Shared language

Even if you don't know F#, you have an idea of what the important concepts are

'|' means a choice -- pick one from the list

```
module CardGame =

    type Suit = Club | Diamond | Spade | Heart

    type Rank = Two | Three | Four | Five | Six | Seven | Eight
                    | Nine | Ten | Jack | Queen | King

    type Card = Suit * Rank
```

* means a pair. Choose one from each

```
    type Hand = Card list
    type Deck = Card list
```

type

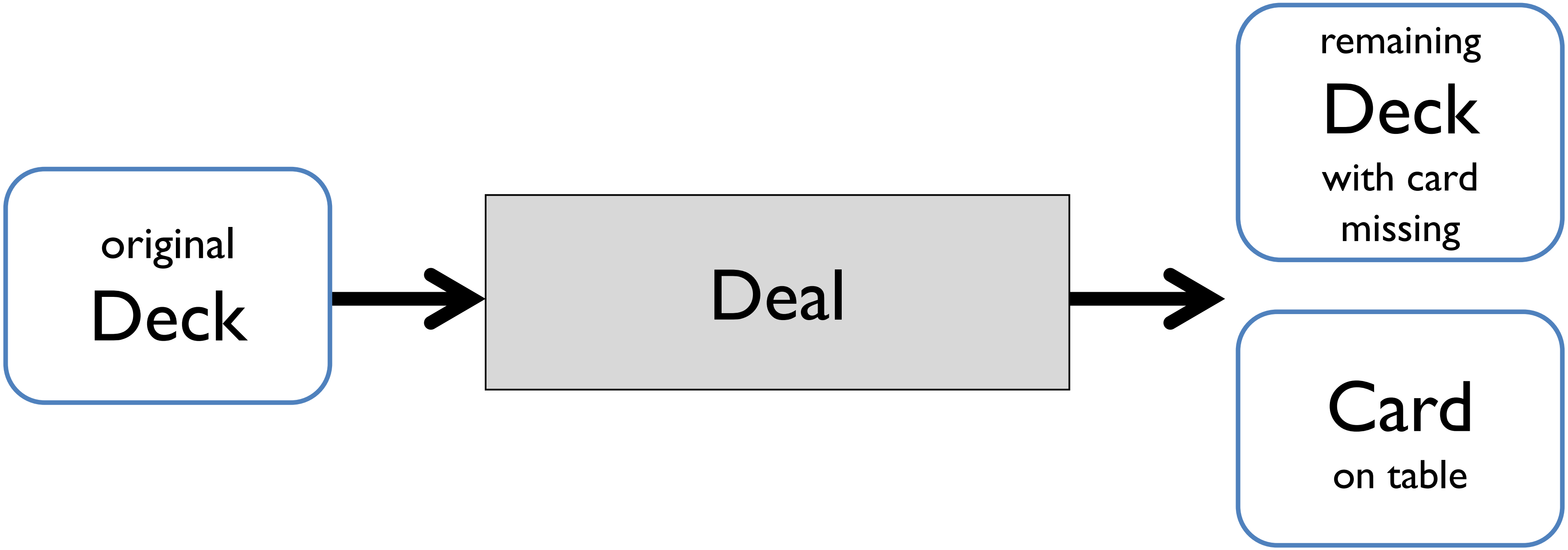
list type is built in

```
    type Player = {Name:string; Hand:Hand}
    type Game = {Deck:Deck; Players: Player list}

    type Deal = Deck -> (Deck * Card)

    type PickupCard = (Hand * Card) -> Hand
```

X -> Y means a workflow

- input of X
- output of Y

Modeling an action with a function



original Deck

Deal

remaining Deck with card missing

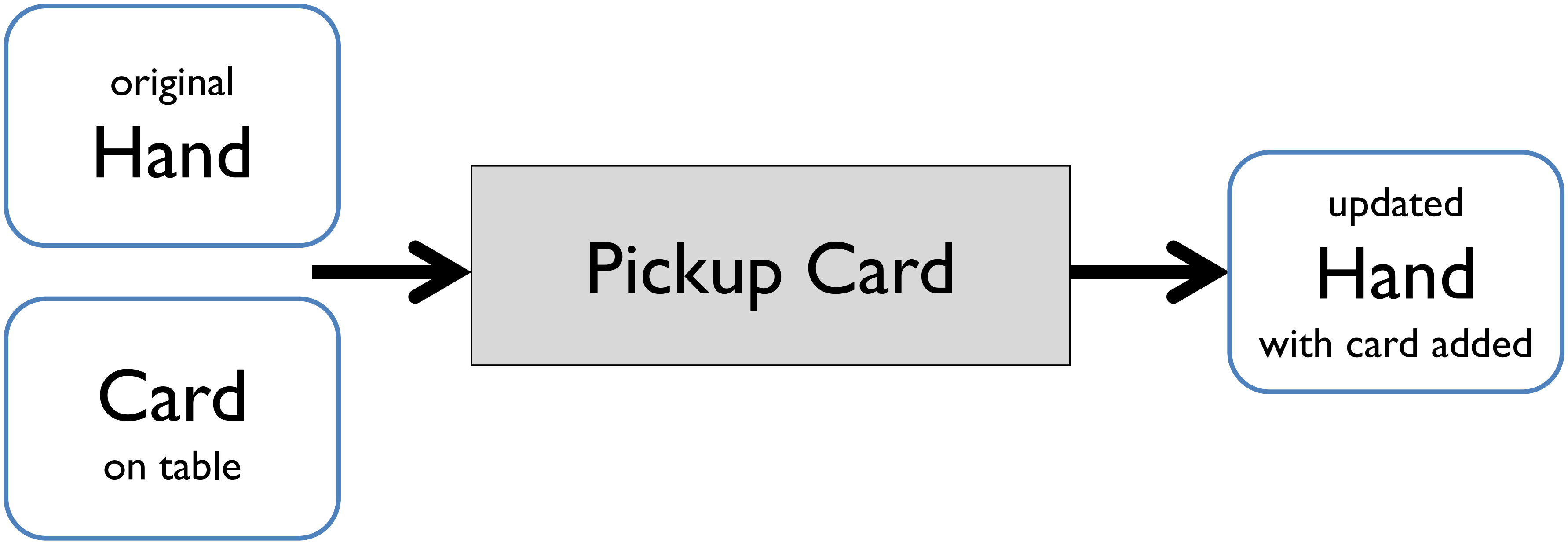
Card on table

```
type Deal = Deck -> (Deck * Card)
```

Input

Output

Modeling an action with a function



```
type PickupCard = (Hand * Card) -> Hand
```

Inpu
t

Outp
ut

```
module CardGame =

    type Suit = Club | Diamond | Spade | Heart

    type Rank = Two | Three | Four | Five | Six | Seven | Eight
                | Nine | Ten | Jack | Queen | King

    type Card = Suit * Rank

    type Hand = Card list
    type Deck = Card list

    type Player = { Name:string;  Hand:Hand }
    type Game = { Deck:Deck;  Players: Player list }

    type Deal = Deck -> (Deck * Card)

    type PickupCard = (Hand * Card) -> Hand
```

Do you think this is a reasonable amount of code to write for this domain?

The whole domain fits on one page!

```
module CardGame =

    type Suit = Club | Diamond | Spade | Heart

    type Rank = Two | Three | Four | Five | Six | Seven | Eight
                  | Nine | Ten | Jack | Queen | King

    type Card = Suit * Rank

    type Hand = Card list
    type Deck = Card list

    type Player = { Name:string;  Hand:Hand }
    type Game = { Deck:Deck;  Players: Player list }

    type Deal = Deck –› (Deck * Card)

    type PickupCard = (Hand * Card) –› Hand
```

Do you think a non-programmer could understand this?

Real comment I heard: "Where's the code?"

```
module CardGame =

  type Suit = Club | Diamond | Spade | Heart

  type Rank = Two | Three | Four | Five | Six | Seven | Eight
                | Nine | Ten | Jack | Queen | King | Ace

  type Card = Suit * Rank

  type Hand = Card list
  type Deck = Card list

  type Player = { Name:string;  Hand:Hand }
  type Game = { Deck:Deck;  Players: Player list }

  type Deal = Deck –› (Deck * Card)

  type PickupCard = (Hand * Card) –› Hand
```

Can non-programmers provide useful feedback?

Anyone spot the mistake?

Rapid feedback during the design stage

Get feedback in minutes rather than days!

```
...
type Deck = Card list
type Deal = Deck -> (Deck * Card)
```

Domain Expert: "This is not right.
We use a shuffled deck to deal"

Me: "So like this? "

```
...

type Deck = Card list
type Deal = ShuffledDeck –› (ShuffledDeck * Card)
```

Expert: "Yes, just like that"

Me: "What's a shuffled deck?

Expert: "It's a list of cards"

...

type **Deck** = Card list

type **Deal** = ShuffledDeck –› (ShuffledDeck * Card)

type **ShuffledDeck** = Card list

Me: "How do you make a shuffled deck?

Expert: "You do a shuffle, duh"

…

type **Deck** = Card list

type **Deal** = ShuffledDeck –› (ShuffledDeck * Card)

type **ShuffledDeck** = Card list

type **Shuffle** = Deck –› ShuffledDeck

```
...
type Deck = Card list
type Deal = ShuffledDeck –› (ShuffledDeck * Card)
type ShuffledDeck = Card list
type Shuffle = Deck –› ShuffledDeck
```

The design process can happen fast and interactively without writing "code"

A side effect is that everyone shares knowledge, and everyone becomes more expert in the domain

Final version of the domain

```
module CardGame =

    type Suit = Club | Diamond | Spade | Heart
    type Rank = Two | Three | Four | Five | Six | Seven | Eight | ...
    type Card = Suit * Rank

    type Hand = Card list
    type Deck = Card list

    type Player = { Name:string;  Hand:Hand }
    type Game = { Deck:Deck;  Players: Player list }

    type Deal = ShuffledDeck –› (ShuffledDeck * Card)
    type ShuffledDeck = Card list
    type Shuffle = Deck –› ShuffledDeck
    type PickupCard = (Hand * Card) –› Hand
```

It's domain-driven,
not database-driven

Nothing about FKs etc

"Persistence ignorance

```
module CardGame =

    type Suit = Club | Diamond | Spade | Heart
    type Rank = Two | Three | Four | Five | Six | Seven | Eight | ...
    type Card = Suit * Rank

    type Hand = Card list
    type Deck = Card list

    type Player = { Name:string;  Hand:Hand }
    type Game = { Deck:Deck;  Players: Player list }

    type Deal = ShuffledDeck –› (ShuffledDeck * Card)
    type ShuffledDeck = Card list
    type Shuffle = Deck –› ShuffledDeck
    type PickupCard = (Hand * Card) –› Hand
```

It's not OO-driven

No base classes, managers, factories, etc.

In the real world	In the code
Suit	Suit
Rank	Rank
Card	Card
Hand	Hand
Deck	Deck
Player	Player
Deal	Deal

The domain code should
be in sync with the
real world vocabulary

In the real world	In the code
Suit	Suit
Rank	Rank
Card	Card
Hand	Hand
Deck	Deck
Player	Player
Deal	Deal
ShuffledDeck	ShuffledDeck
Shuffle	Shuffle

If we learn new things about the domain, the code should reflect that

✓

In the real world	In the code
Suit	Suit
Rank	Rank
Card	Card
Hand	Hand
Deck	Deck
Player	Player
Deal	Deal
	PlayerController
	DeckBase
	AbstractCardProxyFactoryBean

The "domain" code should not use programmer jargon

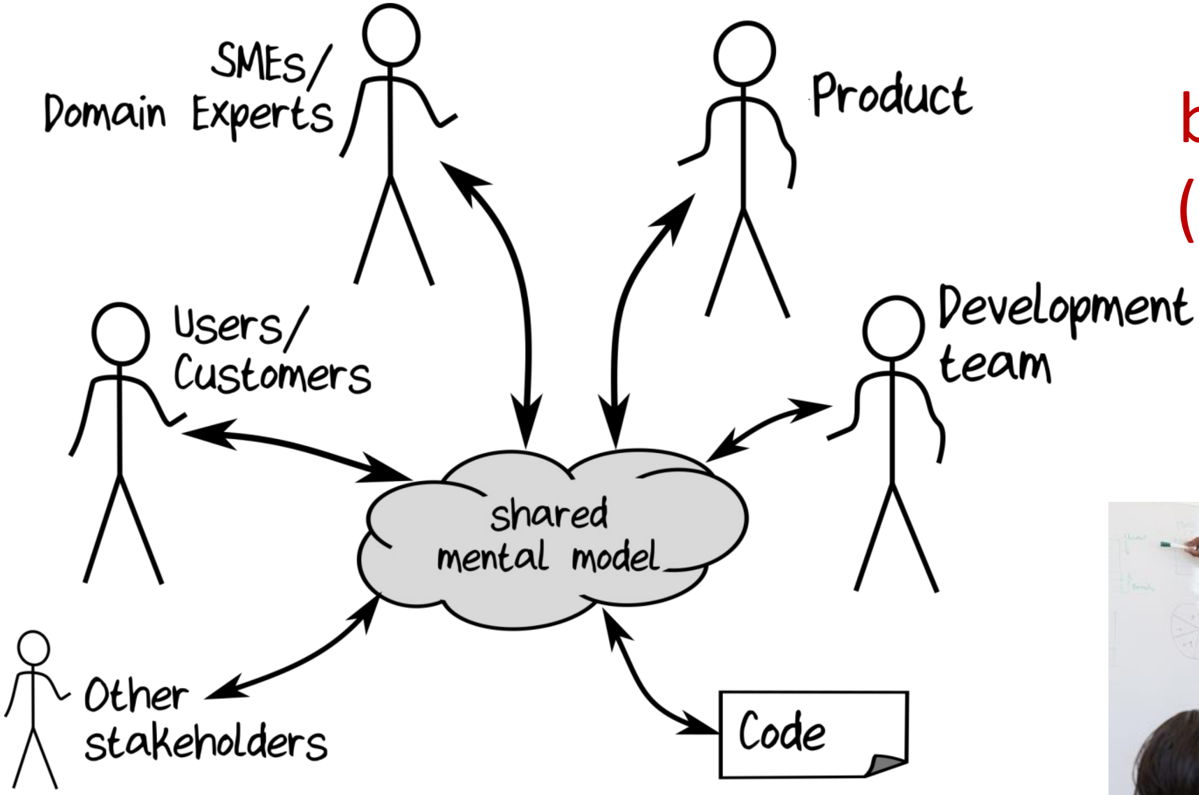
✗

```
module CardGame =

  type Suit = Club | Diamond | Spade | Heart
  type Rank = Two | Three | Four | Five | Six | Seven | Eight | ...
  type Card = Suit * Rank

  type Hand = Card list
  type Deck = Card list

  type Player = { Name:string;  Hand:Hand }
  type Game = { Deck:Deck;  Players: Player list }

  type Deal = ShuffledDeck –› (ShuffledDeck * Card)
  type ShuffledDeck = Card list
  type Shuffle = Deck –› ShuffledDeck
  type PickupCard = (Hand * Card) –› Hand
```

"The design is the code, and the code is the design."

This is not pseudocode – this is executable code!

It's not just about creating a document



The process of becoming an expert (building the shared mental model) is important!

Key DDD principle:

Communicate the design
in the code

A domain modeling challenge!

```
type Contact = {

  FirstName: string
  MiddleInitial: string
  LastName: string

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

Is the design communicated in this code?

```
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 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?

Any domain logic?

Not communicating
the design in the code 😞

```
type Contact = {

  FirstName: string
  MiddleInitial: string
  LastName: string

  EmailAddress: string
  IsEmailVerified: bool
  }
```

Which values are optional?

What are the constraints?

Any domain logic?

Functional domain modeling CAN communicate all these decisions!

Part III
Understanding FP type systems

Why *functional* domain modeling instead of OO domain modeling?

FP principle:
Composition everywhere

Composable

~~Algebraic~~ type system

New types are built from smaller types by:

- Composing with "AND"
- Composing with "OR"

Compose with "AND"

FruitSalad = One each of and and

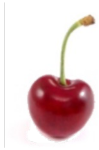
A record type

```
type FruitSalad = {
    Apple: AppleVariety
    Banana: BananaVariety
    Cherry: CherryVariety
    }
```

Another type, the set of all possible apples

All languages have this.
Example: pairs, tuples, records

Compose with “OR”

Snack = or or 

A choice type

```
type Snack =
    | Apple of AppleVariety
    | Banana of BananaVariety
    | Cherry of CherryVariety
```

Again, the set of all possible apples

Not generally available in non-FP languages

A real world example of composing types

Some requirements:

We accept three forms of payment:
Cash, PayPal, or Card.

For Cash we don't need any extra information
For PayPal we need a email address
For Cards we need a card type and card number

How would you implement this?

In OO design you would probably implement it as an interface and a set of subclasses, like this:

```
interface IPaymentMethod
{..}

class Cash() : IPaymentMethod
{..}

class PayPal(string emailAddress): IPaymentMethod
{..}

class Card(string cardType, string cardNo) : IPaymentMethod
{..}
```

In FP you would probably implement by composing types, like this:

```
type EmailAddress = string
type CardNumber = string
```

Primitive types

```
type EmailAddress = ...
type CardNumber = …

type CardType = Visa | Mastercard
type CreditCardInfo = {
   CardType : CardType
   CardNumber : CardNumber
   }
```

Choice type
(using OR)

Record type (using AND)

```
type EmailAddress = ...
type CardNumber = ...
type CardType = ...
type CreditCardInfo = ...

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

Choice type

```
type EmailAddress = ...
type CardNumber = ...
type CardType = ...
type CreditCardInfo = ...

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

Information associated with that choice

```
type EmailAddress = ...
type CardNumber = ...
type CardType = ...
type CreditCardInfo = ...
type PaymentMethod =
  | Cash
  | PayPal of EmailAddress
  | Card of CreditCardInfo

type PaymentAmount = decimal
type Currency = EUR | USD
```

Another primitive type

Another choice type

```
type EmailAddress = ...
type CardNumber = ...
type CardType = ...
type CreditCardInfo = ...
type PaymentMethod =
  | Cash
  | PayPal of EmailAddress
  | Card of CreditCardInfo
type PaymentAmount = decimal
type Currency = EUR | USD

type Payment = {
  Amount : PaymentAmount
  Currency : Currency
  Method : PaymentMethod   }
```

Record type

```
type EmailAddress = ...
type CardNumber = ...
type CardType = ...
type CreditCardInfo = ...
type PaymentMethod =
  | Cash
  | PayPal of EmailAddress
  | Card of CreditCardInfo
type PaymentAmount = decimal
type Currency = EUR | USD
```

```
type Payment = {
  Amount : PaymentAmount
  Currency : Currency
  Method : PaymentMethod  }
```

Final type built from many smaller types:

Composition ftw!

Part IV
Domain modeling with composable types

Let's apply this approach to the "contact" challenge

```
type Contact = {

  FirstName: string
  MiddleInitial: string
  LastName: string

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

This looks suspiciously like database-driven design...

Let's refactor it to make it domain-driven!

"A contact has a name AND email info"

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

"Like this?..."

"A contact has a name AND email info"

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

We have two new concepts already!

"What's a personal name?"

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

"What's required or optional?"

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

required

optional

required

How can we represent optional values?

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

nice and readable!

Modeling simple values and constrained values

Modeling simple values

- Avoid "Primitive Obsession"
- Simple values should not be modelled with primitive types like "int" or "string" or "float"

"Does 'float' have something to do with water?"

Modeling constrained values

- It's rare to have an unconstrained int or string:
  - An EmailAddress must not be empty,
    it must match a pattern
  - A PhoneNumber must not be empty,
    it must match a pattern
  - A CustomerId must be a positive integer

Is an EmailAddress just a string? No!

Is a CustomerId just a int? No!

Can you concat two EmailAddresses
to make another valid EmailAddress?

Can you multiply a CustomerId by 42?

Use wrapper types to keep domain concepts distinct from their representation

Wrap a string

```
type EmailAddress = EmailAddress of string
type CustomerId = CustomerId of int
```

Wrap an int

A constrained string

```
type String50 = String50 of string
type EmailAddress = EmailAddress of string
type PhoneNumber = PhoneNumber of string
```

Distinct types

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

Also distinct types

Two benefits:

- Clearer domain modelling
- Can't mix them up accidentally

Implementing constructors for constrained types

```
let createEmailAddress (s:string) =
  if s.Contains("@")
    then (EmailAddress s)
    else ?
```

```
createEmailAddress:
  string -> EmailAddress
```

This is a lie!

```
let createEmailAddress (s:string) =
  if s.Contains("@")
    then Some (EmailAddress s)
    else None
```

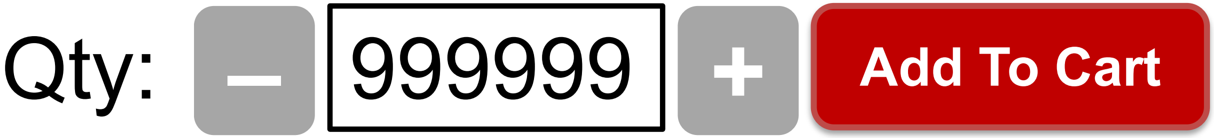
```
createEmailAddress:
  string -> EmailAddress option
```

↑

This is more explicit

```
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 wrapper 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
```

The "Contact" challenge, after first refactor

```
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
  }
```

Use option type for potentially missing values

Use wrapper types instead of primitives

```
type Contact = {

  FirstName: String50
  MiddleInitial: String1 option
  LastName: String50

  EmailAddress: EmailAddress
  IsEmailVerified: bool
  }
```

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

2 different domain concepts

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

Aggregates a.k.a. "consistency boundaries"

Replacing flags with choices

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

What about this?

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

But anyone can set this to true

- *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*

Listen closely to what the domain expert says...

"An email address is either Verified OR Unverified"

So model it as a choice

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

"Email contact info is either Verified OR Unverified"

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

Use a normal email address

"Email contact info is either Verified OR Unverified"

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

Let's ask a domain expert!

Q: Is a verified email different from an unverified email? Are there different business rules?

A: Yes, it must not be mixed up with unverified.

```
type VerifiedEmail = VerifiedEmail of EmailAddress
```

Create a wrapper for a verified email address

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

"Email contact info is either Verified OR Unverified"

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

A different type!

Q: Where do we get Verified emails from?

A: A special verification process

So model it as a function

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

You give me this

And I *might* give you this

Q: Are the business rules clear now?

```
type VerifiedEmail =
  VerifiedEmail of EmailAddress

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

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

Q: Are the business rules clear now?

```
type VerifiedEmail =
  VerifiedEmail of EmailAddress

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

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

To create the unverified case, you can use any email address

Q: Are the business rules clear now?

```
type VerifiedEmail =
  VerifiedEmail of EmailAddress

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

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

To create this case, you need to have a VerifiedEmail

```
type VerifiedEmail =
  VerifiedEmail of EmailAddress

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

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

The only way to get a VerifiedEmail is to use the verification service!

Those business rules are automatically enforced by the design!

The "Contact" challenge, completed

Before redesign

```
type Contact = {

    FirstName : string
    MiddleInitial : string
    LastName : string

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

Looks suspiciously like database-driven design

After redesign

```
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 domain language is evolving along with the design

And all this is compilable code, of course

Refactoring towards deeper insight

Refactoring towards deeper insight

Business rule: Only send password resets to verified emails

```
let sendPasswordReset (info:EmailContactInfo) =
// if EmailContactInfo.IsVerified then
  //    logic to send password reset
  // else
  //    error
```

Unclear what the logic is.
We need to look at the documentation

Refactoring towards deeper insight

Business rule: Only send password resets to verified emails

```
type VerifiedEmail = ...
```

We learned a new concept which is applicable to other workflows

Refactoring towards deeper insight

Business rule: Only send password resets to verified emails

```
type VerifiedEmail = ...


 let sendPasswordReset (email:VerifiedEmail) =
   // logic to send password reset
   //   (boolean test no longer needed)
```

Better documentation and safer too!

Part V
Encoding business rules with types

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

New! Added as the design evolves.

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

and note the use of "OR"

"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!

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

Only <u>three</u> possibilities. You cannot make a mistake.

Collaboration is two-way.
It's OK to push back

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

Is this really what the business wants?

This implementation is too rigid

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

Better

"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

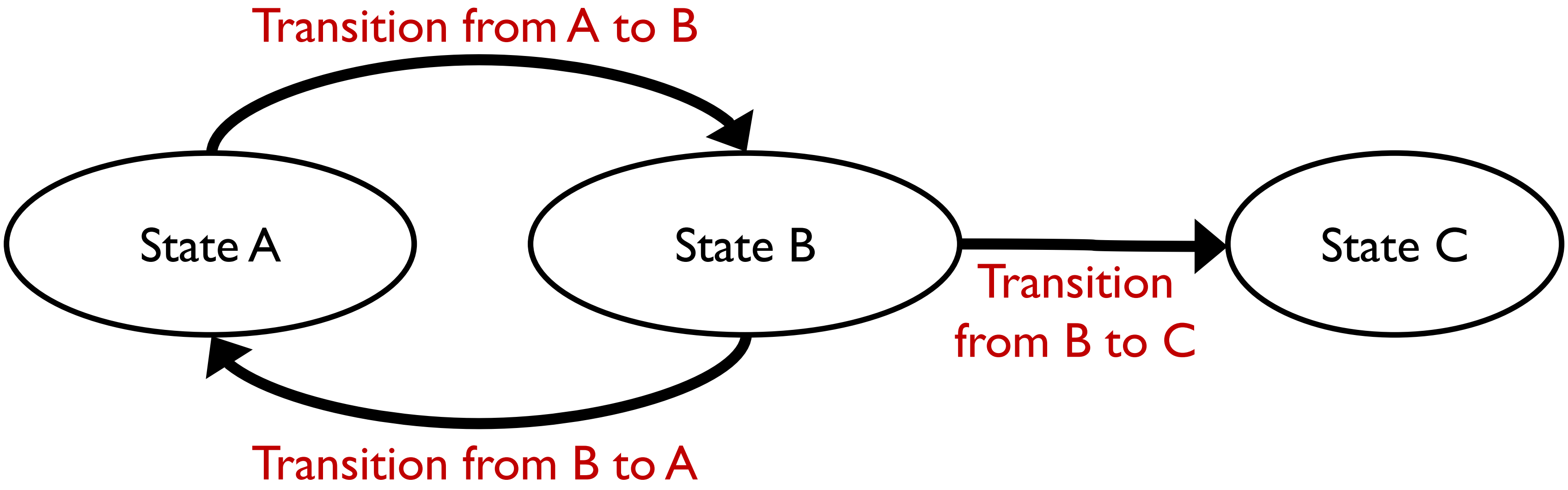
This design is better because it can evolve more easily

```
type ContactInfo =
    | Email of EmailContactInfo
    | Addr of PostalContactInfo
    | Facebook of FacebookInfo
    | SMS of PhoneNumber
    | Twitter of TwitterId
    | Skype of SkypeId
```

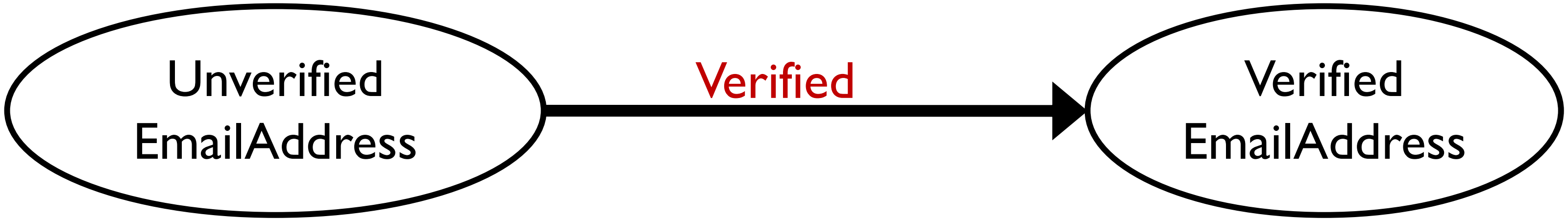
Part VI:
States and Transitions

Modelling a common scenario

States and transitions



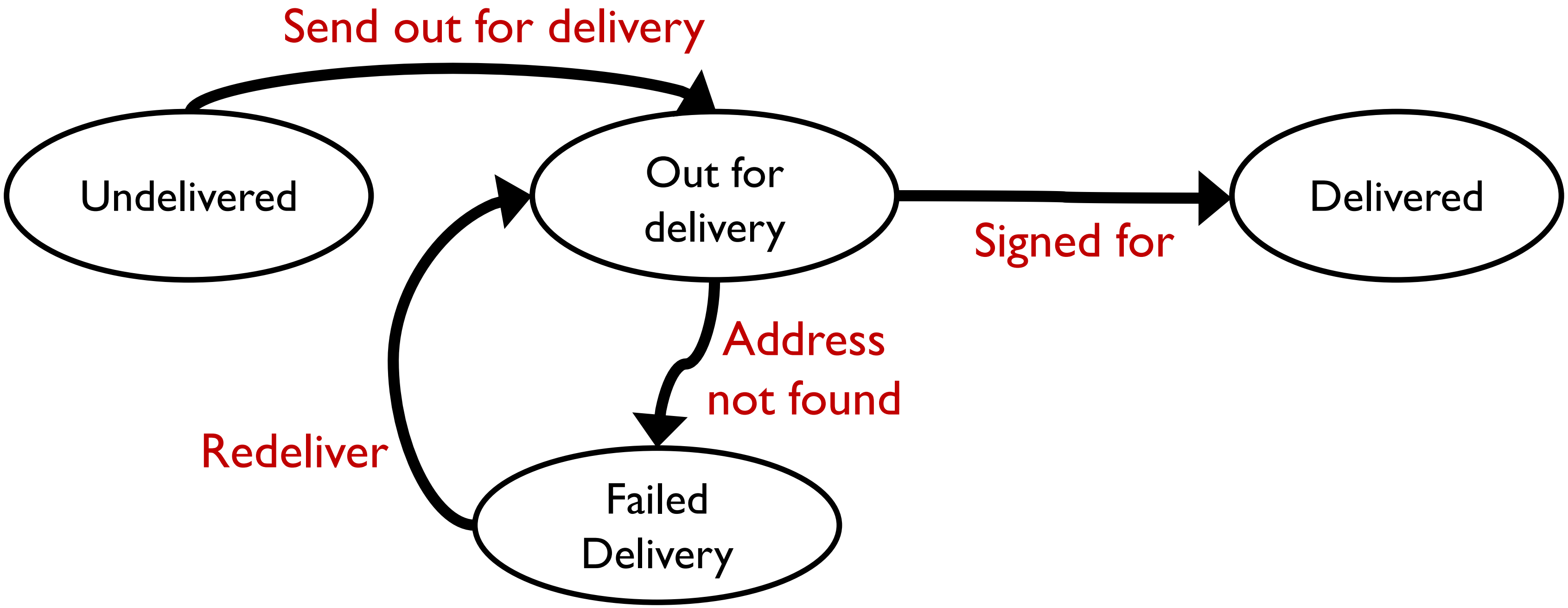
States and transitions for email address



Each state can have different behavior

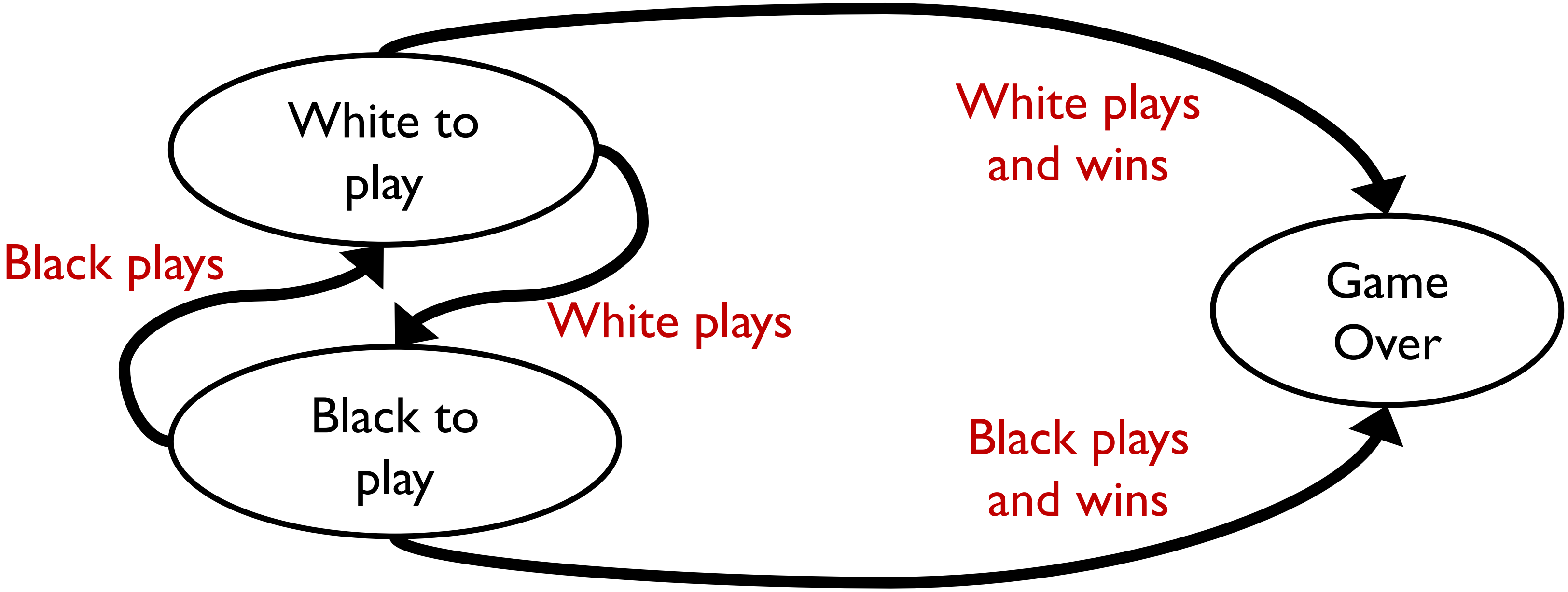
Rule: "You can only send a verification message to an unverified email"
Rule: "You can only send a password reset message to a verified email "

States and transitions for deliveries



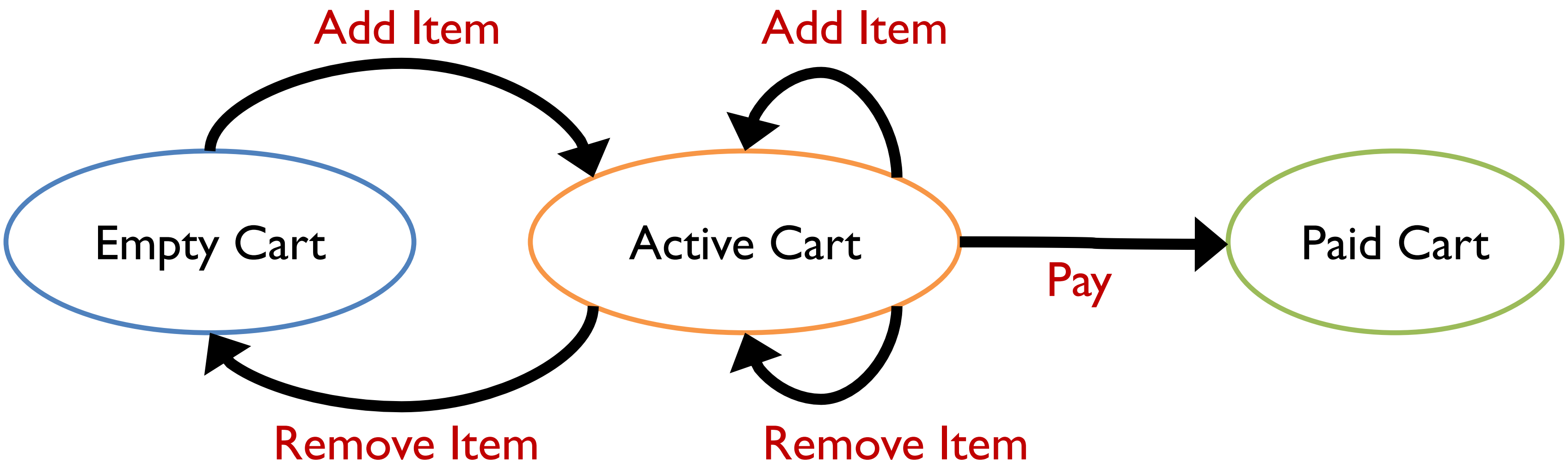
Rule: "You can't put a package on a truck if it is already out for delivery"
Rule: "You can't sign for a package that is already delivered"

States and transitions for chess game



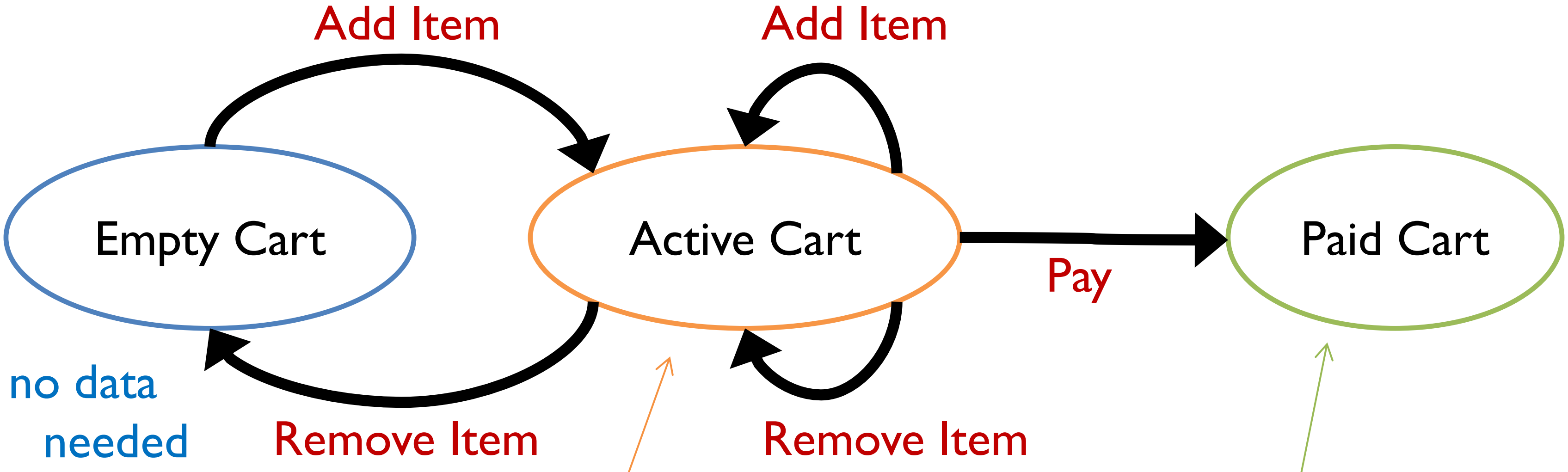
Rule: "White and Black take turns playing.
White can't play if it is Black's turn and vice versa"
Rule: "No one can play when the game is over"

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"

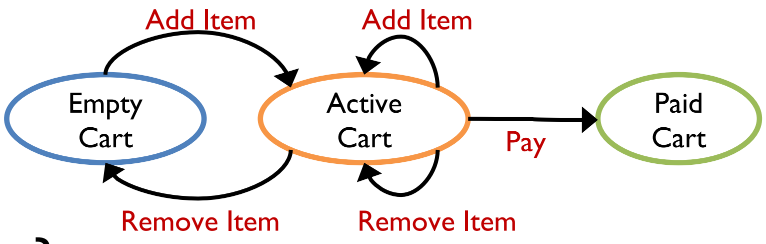
States and transitions for shopping cart



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

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

What data do we need to store?

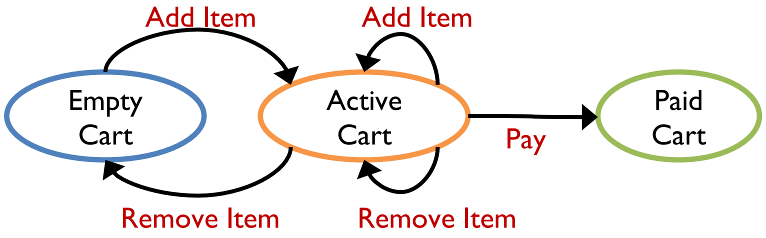


```
type ActiveCartData =
   { UnpaidItems: Item list }
type PaidCartData =
   { PaidItems: Item list; Payment: Payment}


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

pay:
   (ActiveCartData * Payment) -> ShoppingCart
```

might be empty or active – can't tell

Client code to add an item using the API

```
let addItem cart item =
    match cart with
    | EmptyCart ->
        Api.initCart item
    | ActiveCart activeData ->
        Api.addToActive(activeData,item)
    | PaidCart paidData ->
        Api.???
```

Cannot accidentally alter a paid cart! "make illegal operations impossible"

Client code to remove an item using the API

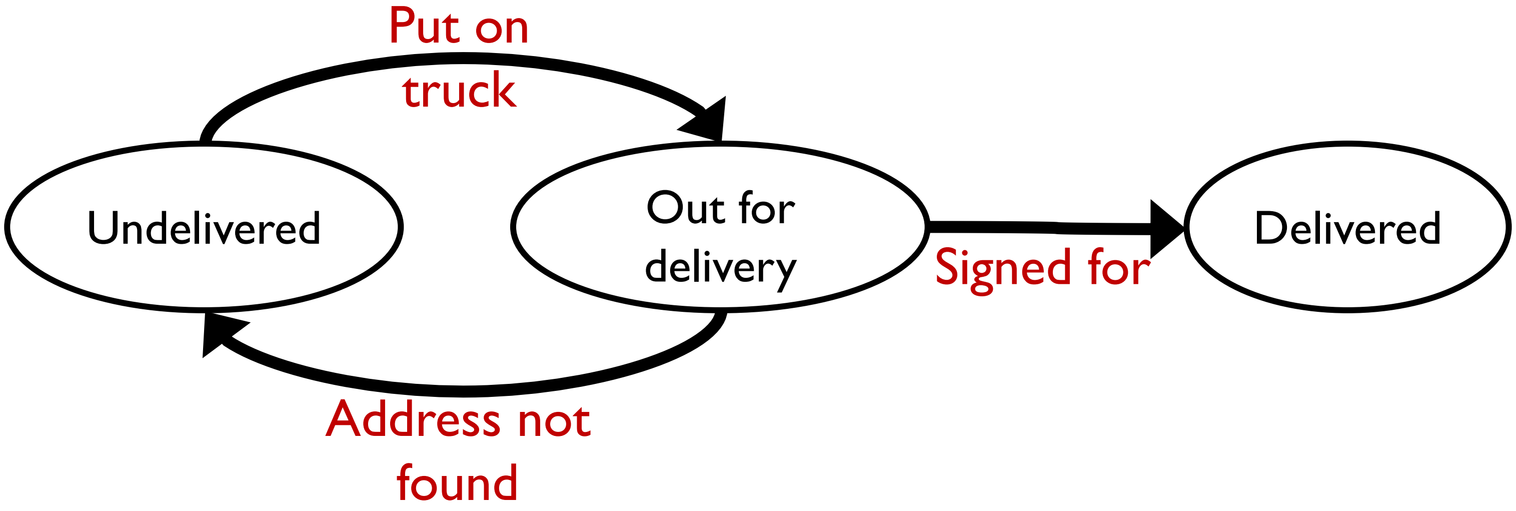
```
let removeItem cart item =
    match cart with
    | EmptyCart ->
        ???
    | ActiveCart activeData ->
       Api.removeFromActive(activeData,item)
    | PaidCart paidData ->
         ???
```

Compiler will not let you remove from an empty cart!

"make illegal operations impossible"

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.



Summary and Conclusion

Domain-driven design

- Represent the shared mental model in code
  - The developers should become domain experts too
  - Write code collaboratively to build the shared mental model
- Designs will evolve
  - Embrace change. This is not Big Design Up Front
  - Refactor towards deeper insight
  - Static types give you confidence to make changes

Coding Guidelines

- Build a domain model with composable types
- Use choices rather than inheritance
- Use constrained types
- Avoid boolean flags
- Make illegal states unrepresentable
- Use state machines
- Use total functions and explicit contracts

All these approaches improve documentation and make it harder to have errors