Domain-Driven Design and Domain Modelling

Why is it important,
and why should you care?

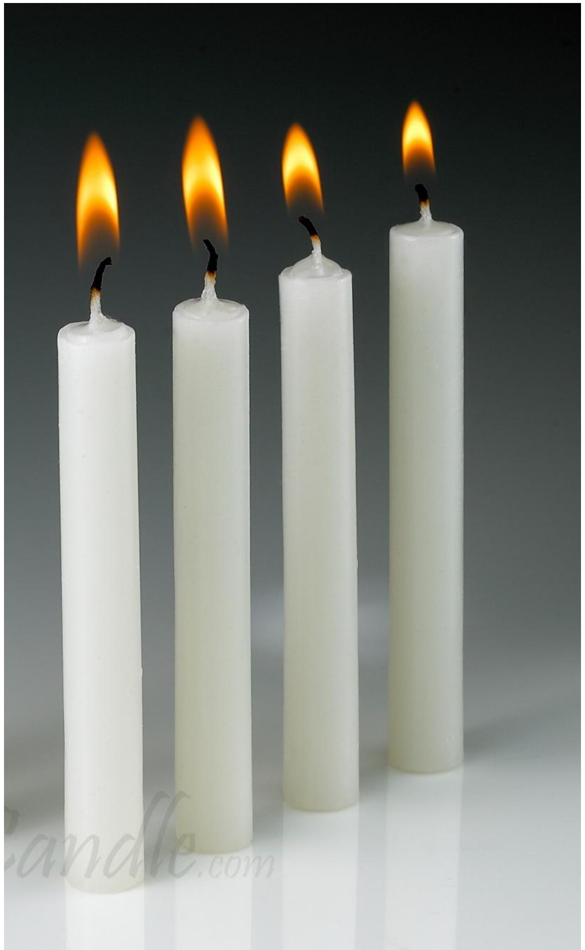
Part I

Communication & Feedback

Let's start with a comedy sketch (famous in UK)

Four candles?

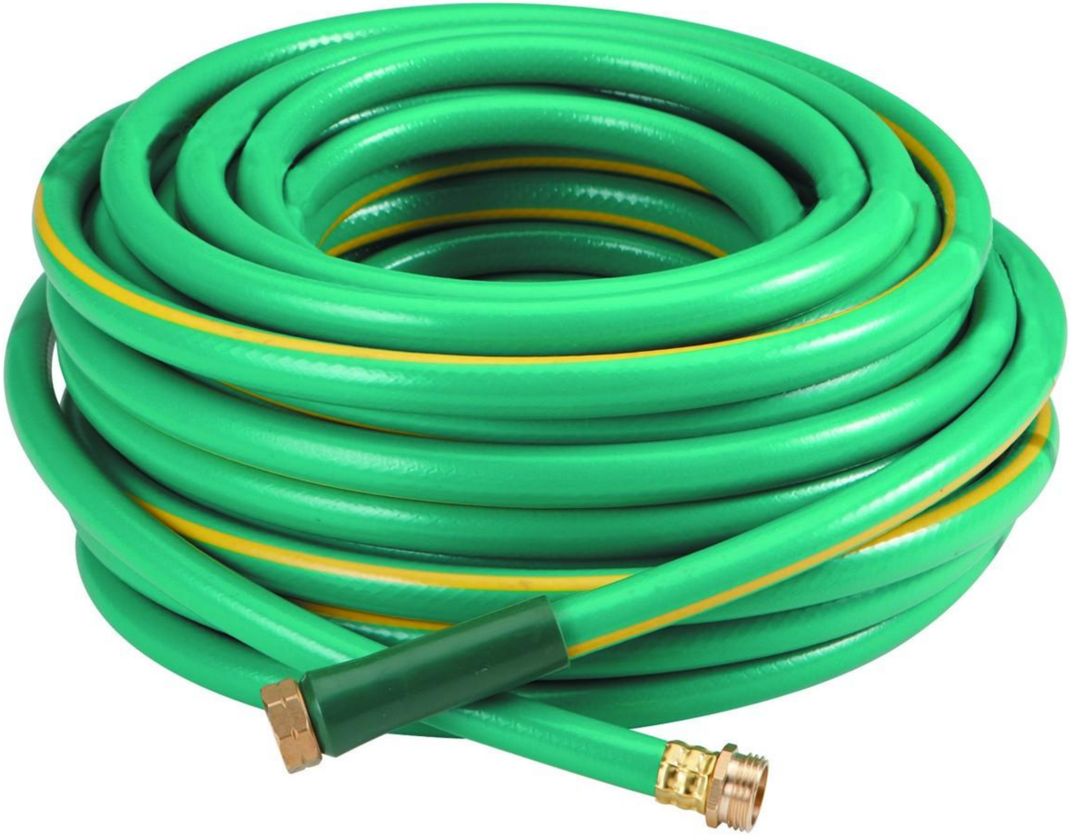
What he thought he heard



What was actually asked for

Some 'ose?

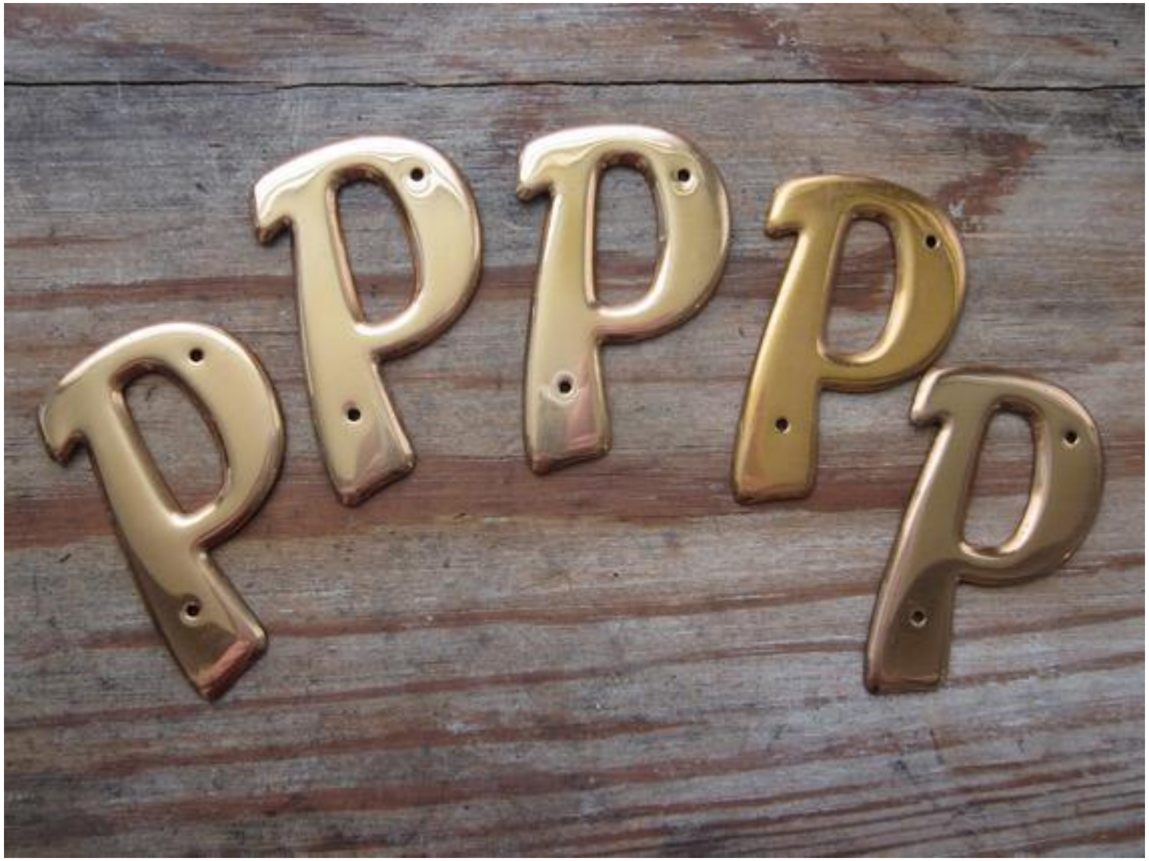
What he thought he heard



What was actually asked for

Some 'p's?
Ok, just like those 'o's

What he thought he heard



What was actually asked for

What's the problem?

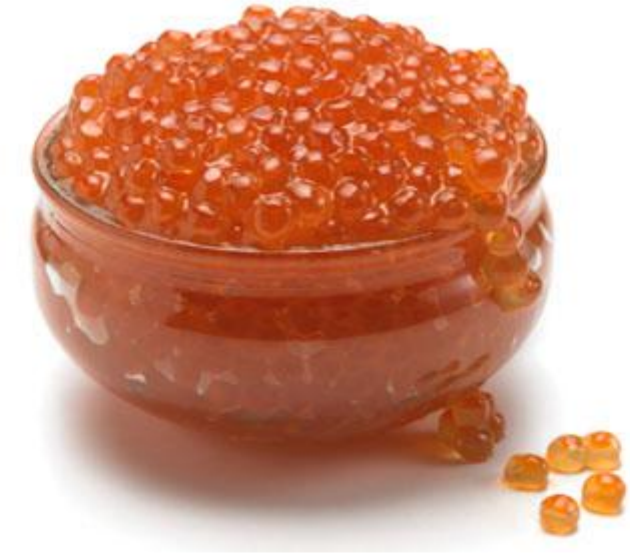
- Misunderstanding the requirement.
- Acting on the requirement without getting feedback first.

Most romantic comedies are based on the same premise.

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

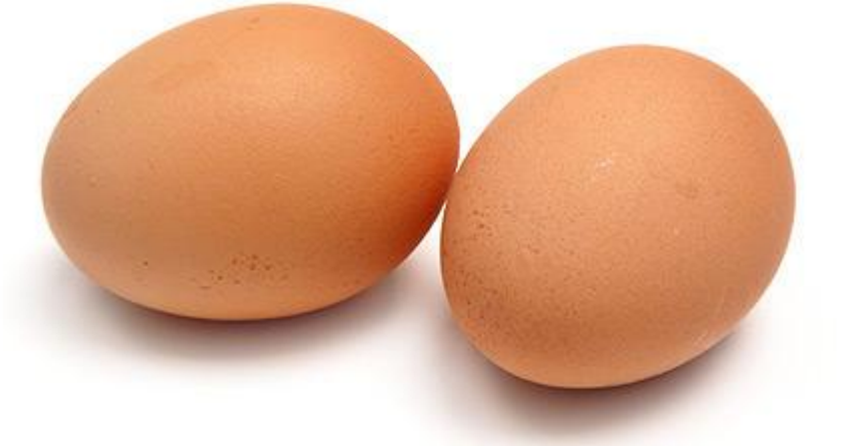
A café example

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



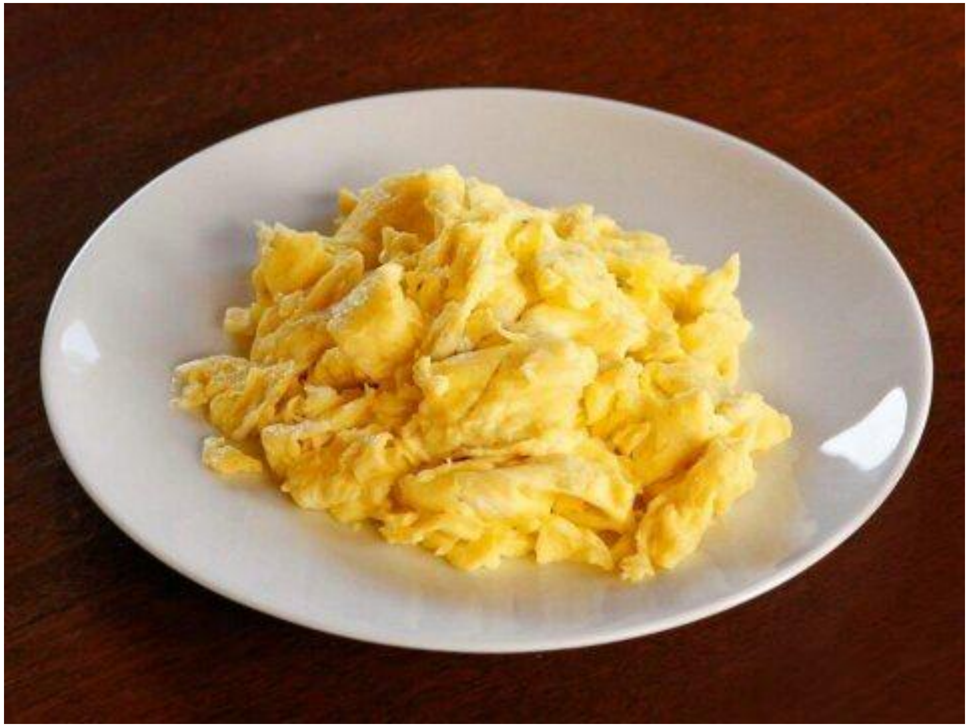
A café example

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



A café example

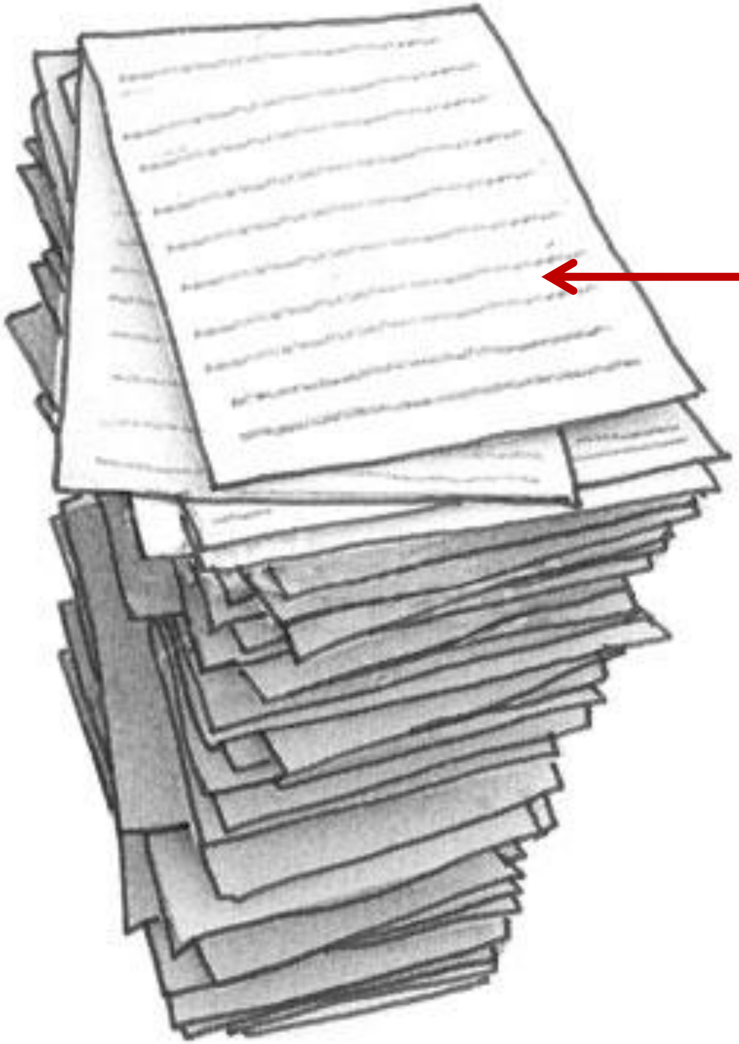
- Waiter to chef: " No, *cooked* chicken eggs"
- Chef: "Ok, this time I understand…"



A café example

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

What's the solution?



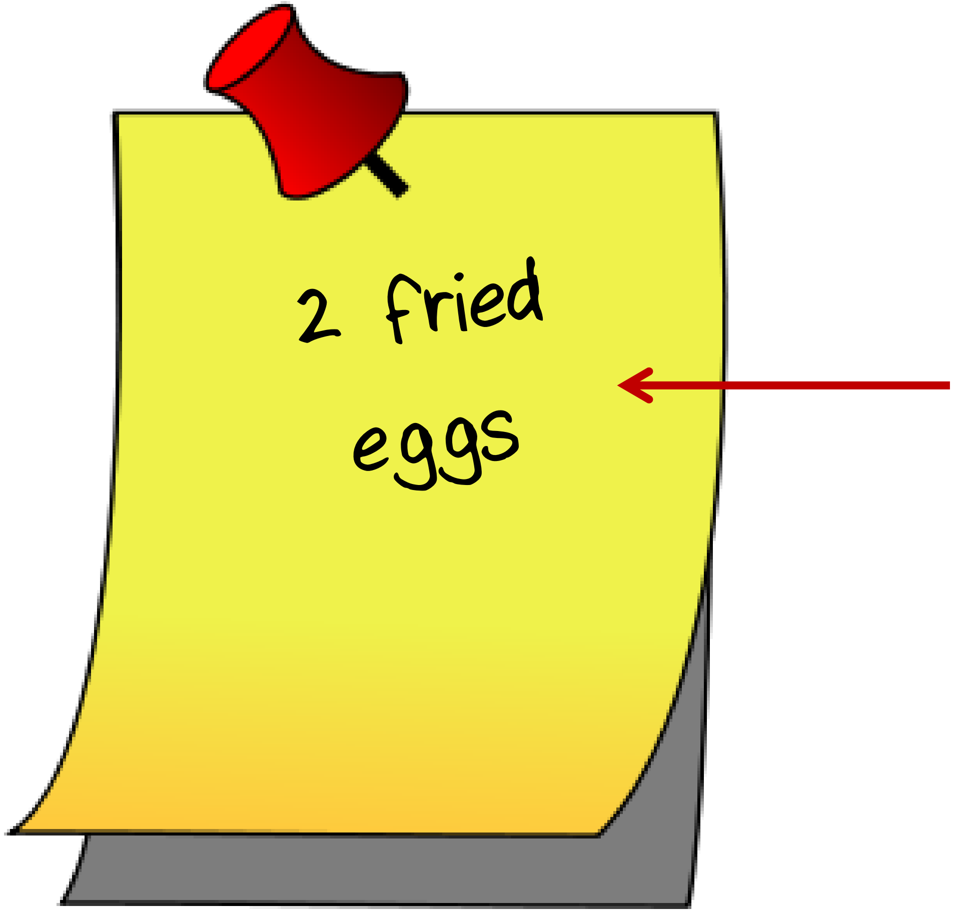
A 200 page spec on how to cook a fried egg

Who thinks this will work?

This is not the solution!

- We expect the chef to be a subject matter expert.
- A 200 page spec should not be needed!

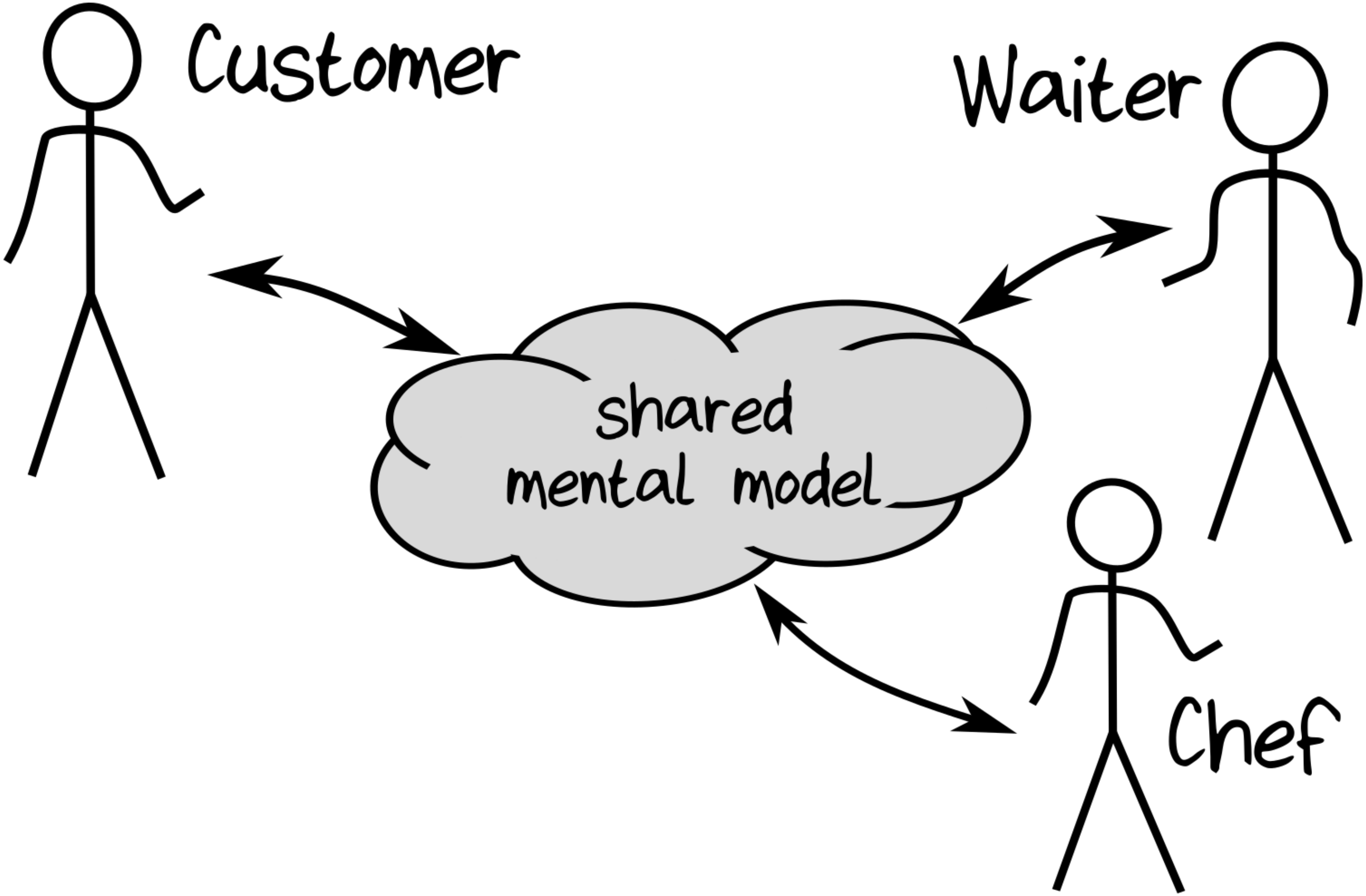
What happens in real life?



A 3 word spec

Why does this tiny spec work?

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


Customer
Waiter
shared
mental model
Chef

Food for thought

- Is this the most minimal spec?
  - What happens if you make eggs for yourself?
  - You don't write anything down!
- What if you visit often?
  - Can you ask for "the usual"?
  - If your colleague makes you tea/coffee, do they know how you like it?

The importance of feedback

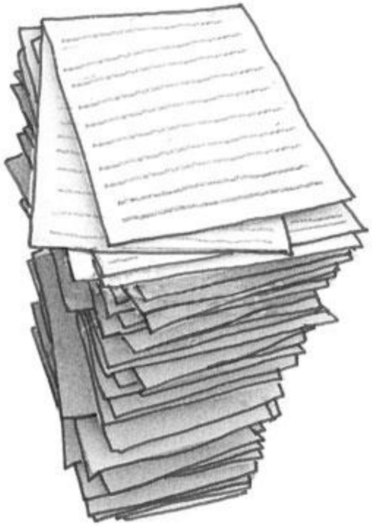
- Fast feedback from the customer is important!
  - The customer can be unclear
  - The waiter can misunderstand
  - The chef can mess up

The importance of feedback

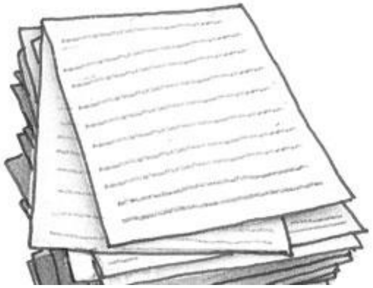
- What if we ordered 2000 eggs for delivery in 3 months time?
  - Do we wait for feedback then?
  - No! Get the customer to taste a sample ASAP! ("needs more salt")

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?

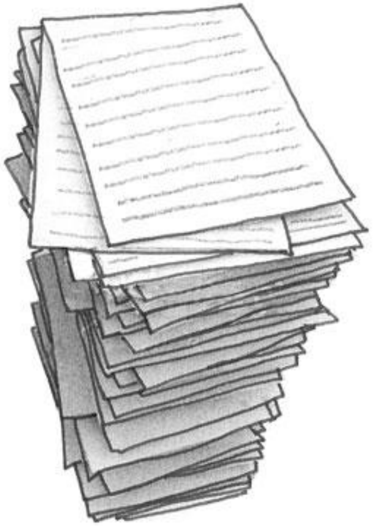


or?

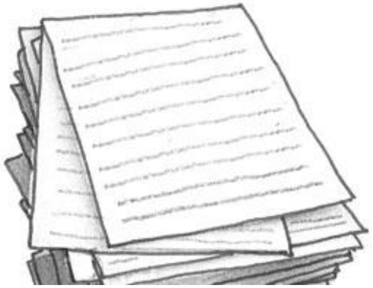


How long must a spec be?

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

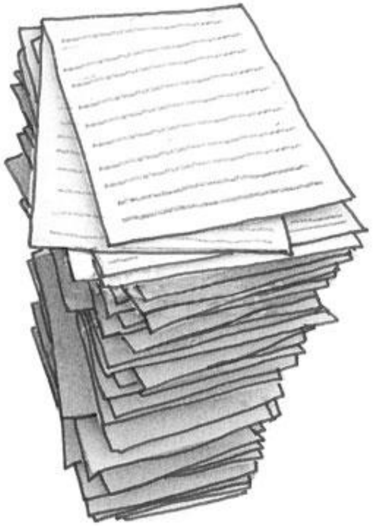


or?

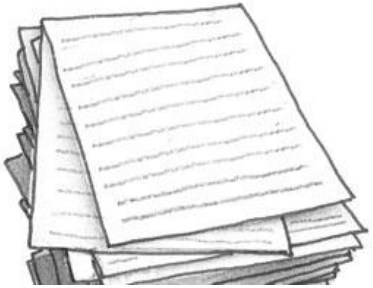


How long must a spec be?

- Which of the two projects is more likely to succeed? Expert or non-expert?



or?



Part II

Efficiency vs. Effectiveness

People like to talk about efficiency a lot

This is an efficient light bulb!

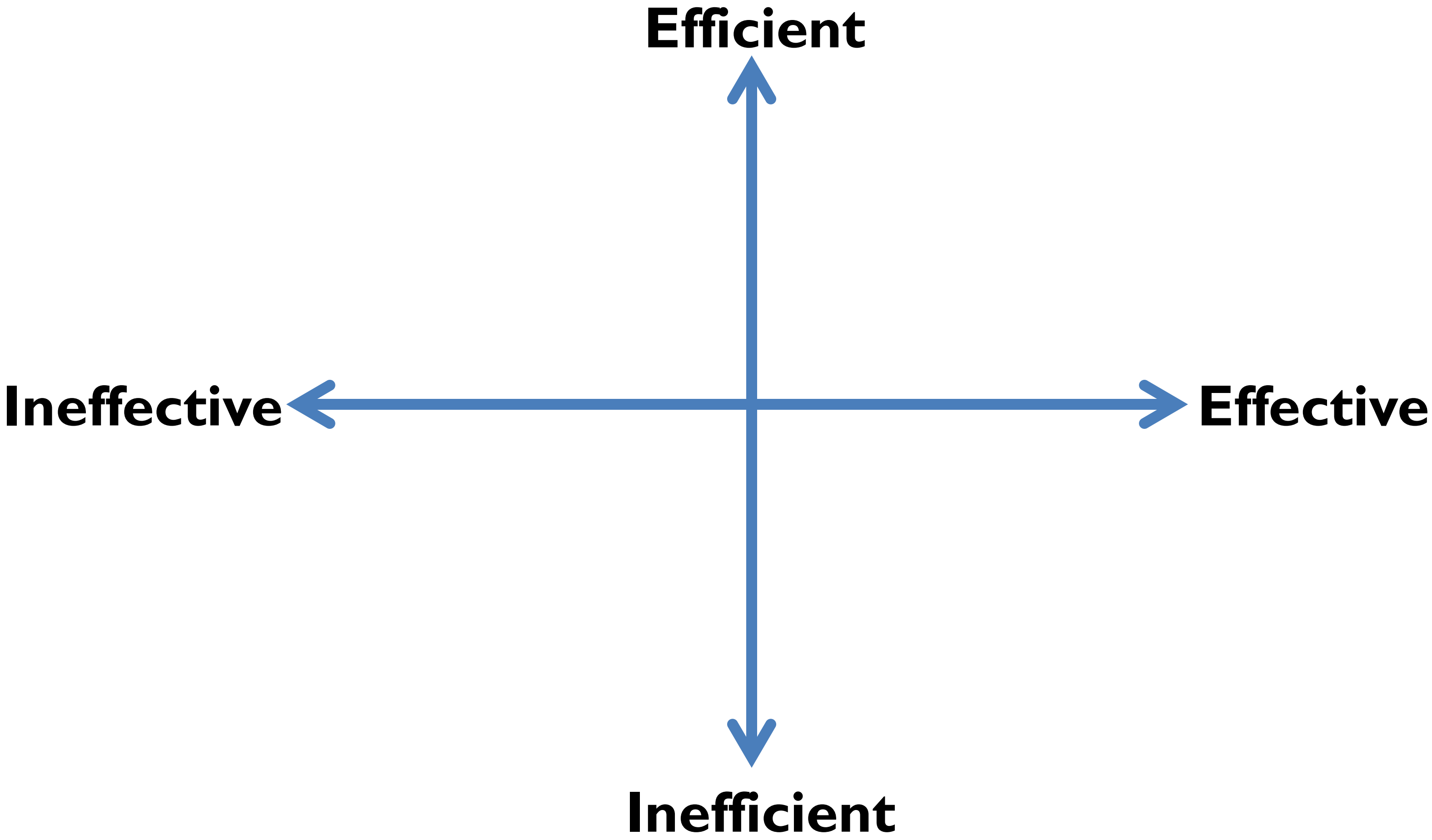
We should really focus on effectiveness

Effectiveness is about the direction, not the speed!

Better to go in the right direction slowly than the wrong direction quickly.

Effectiveness is about the direction, not the speed!

The "right" direction may change,
so check your compass bearing often!



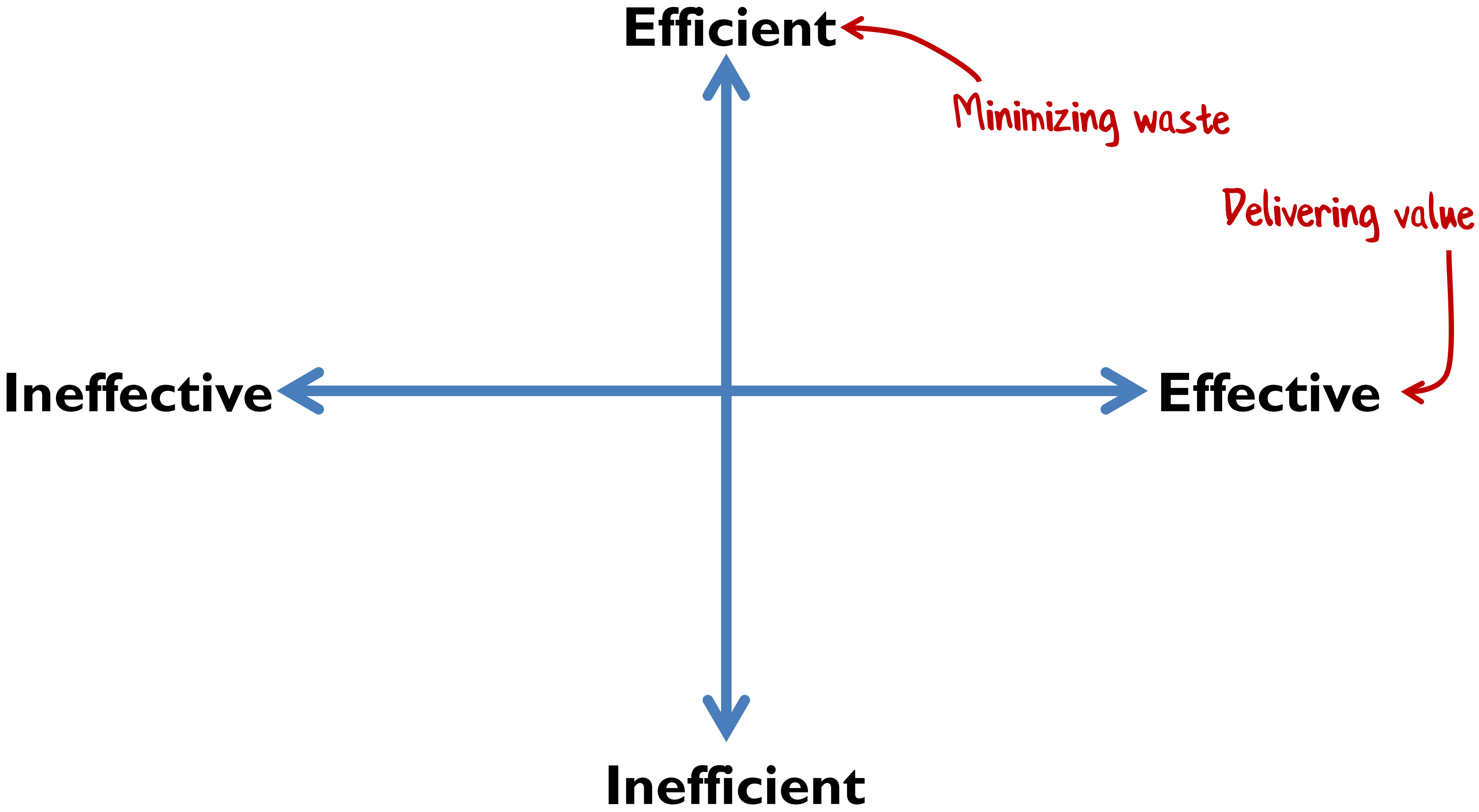
Efficient

Ineffective

Effective

Inefficient

A helpful and un-ironic four-quadrant diagram



Efficient

Minimizing waste

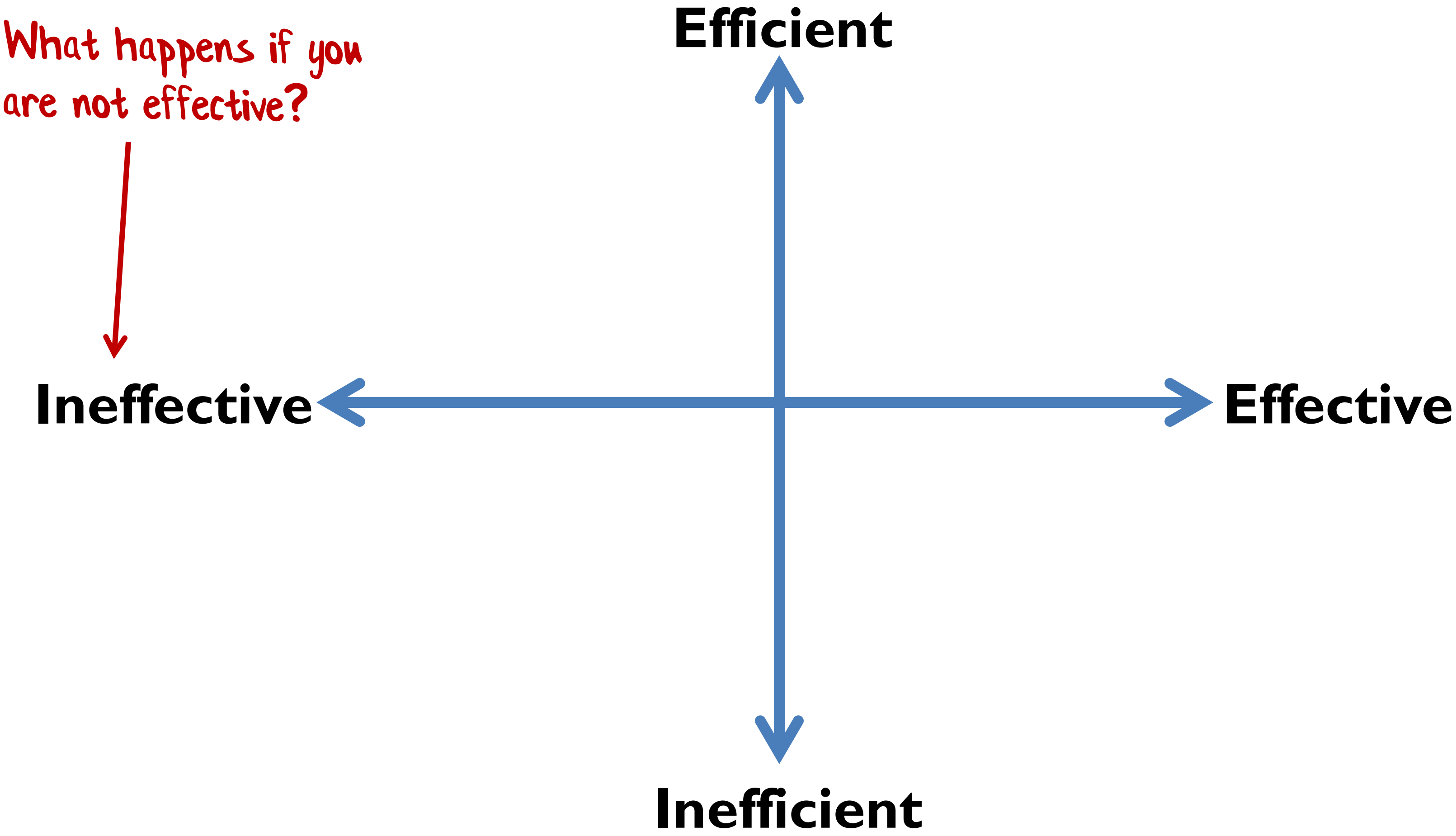
Delivering value

Ineffective

Effective

Inefficient

A helpful and un-ironic four-quadrant diagram



Efficient

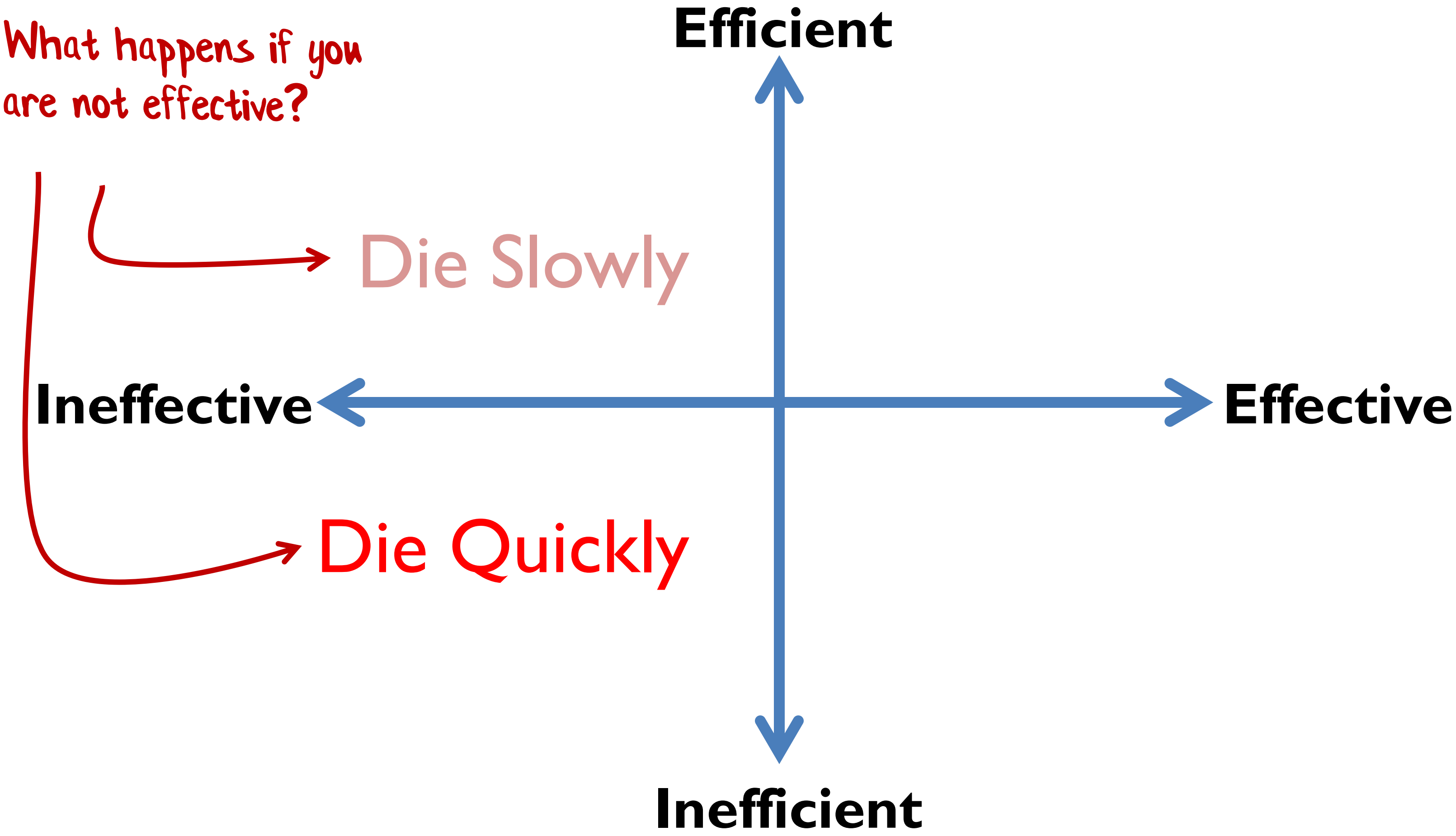
Ineffective

Effective

Inefficient

What happens if you are not effective?

A helpful and un-ironic four-quadrant diagram



What happens if you are not effective?

Efficient

Die Slowly

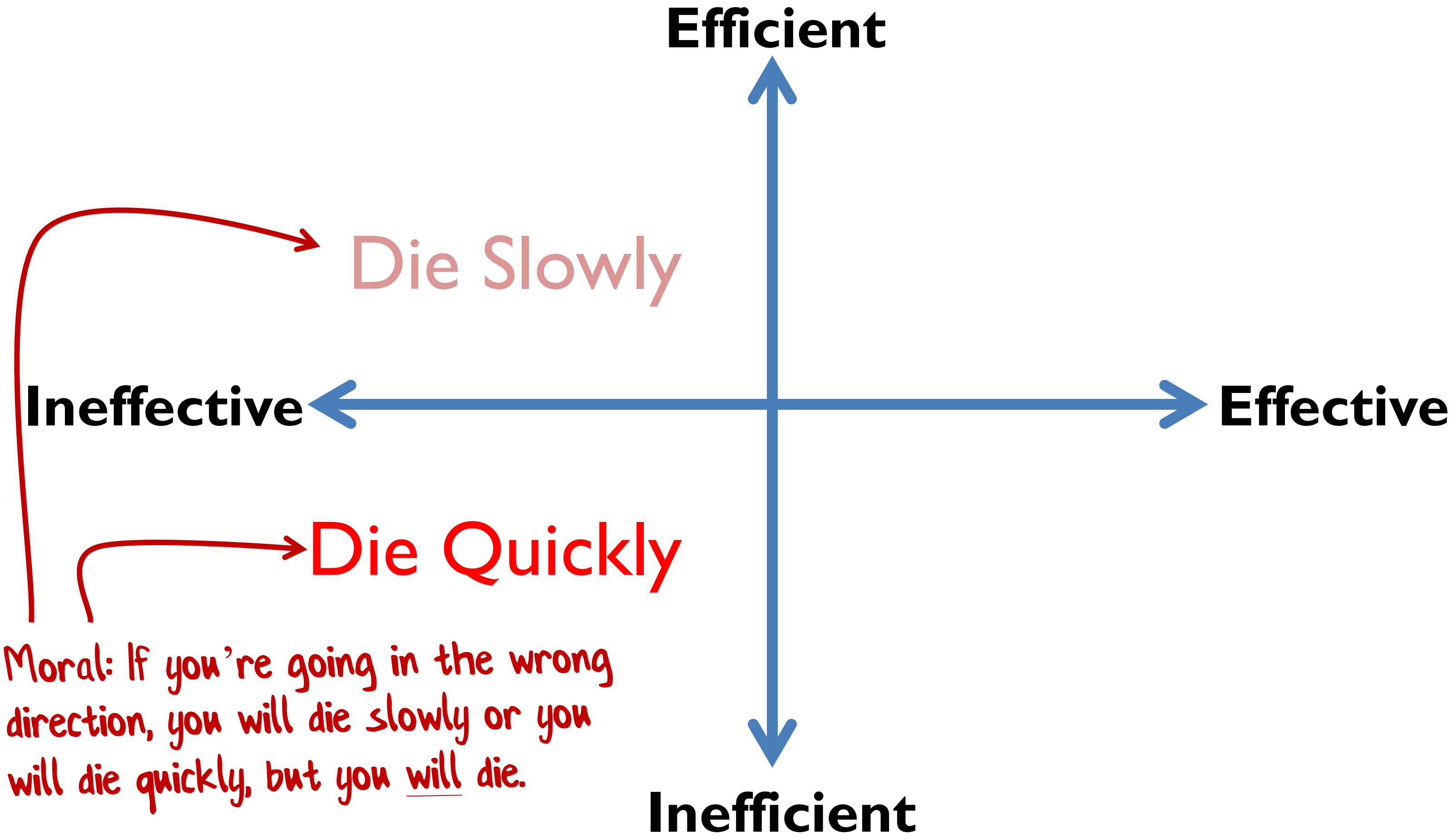
Ineffective

Effective

Die Quickly

Inefficient

A helpful and un-ironic four-quadrant diagram



Efficient

Ineffective

Effective

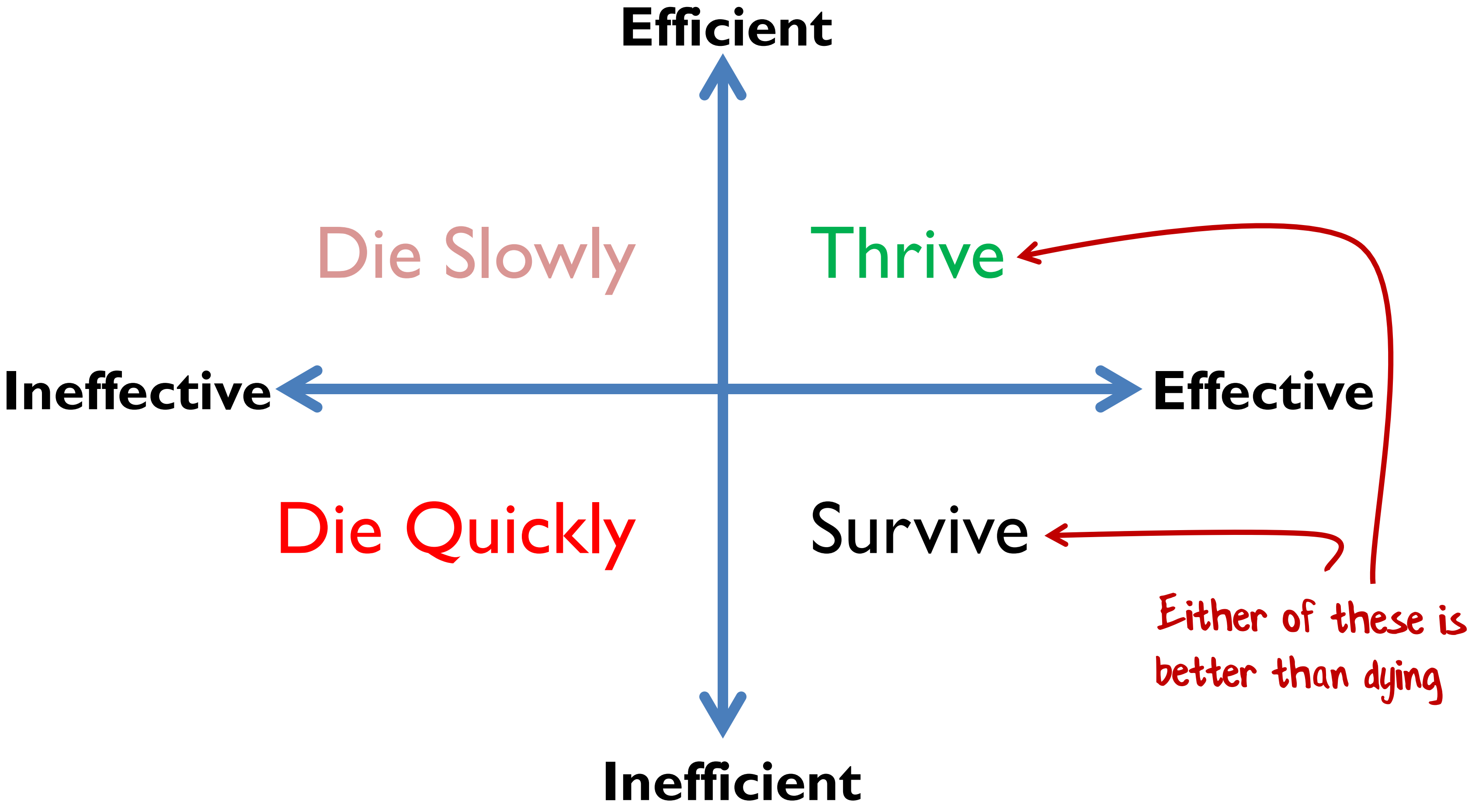
Inefficient

Die Slowly

Die Quickly

Moral: If you're going in the wrong direction, you will die slowly or you will die quickly, but you will die.

A helpful and un-ironic four-quadrant diagram



A helpful and un-ironic four-quadrant diagram

Efficiency is
doing things right

Effectiveness is
doing the right things

Summary so far

- Strive for effectiveness over efficiency
  - Direction is more important than speed
- Communication is easiest with a shared mental model
  - And a shared vocabulary
- Fast feedback is important
  - Check your compass frequently

Part III

Domain Driven Design

What does all this have to do with software projects?

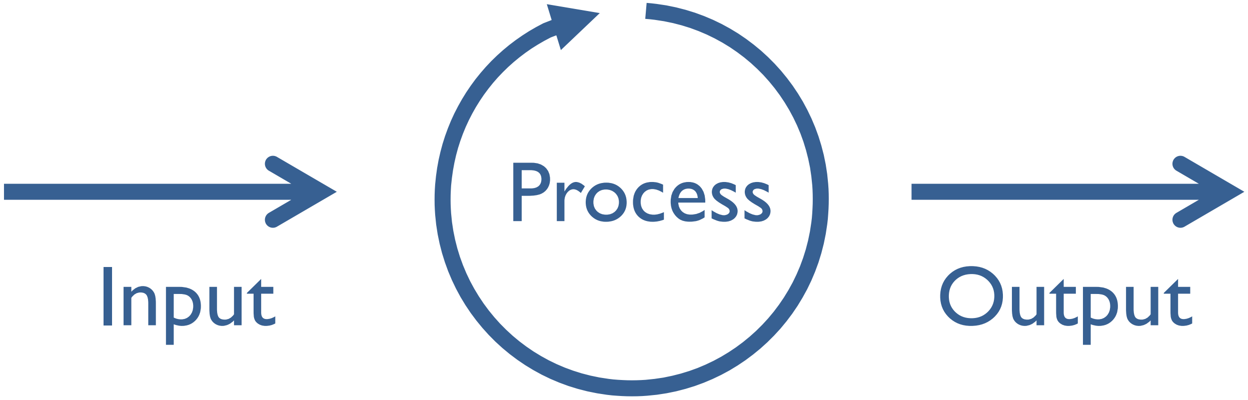
In my experience, most projects fail because:

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

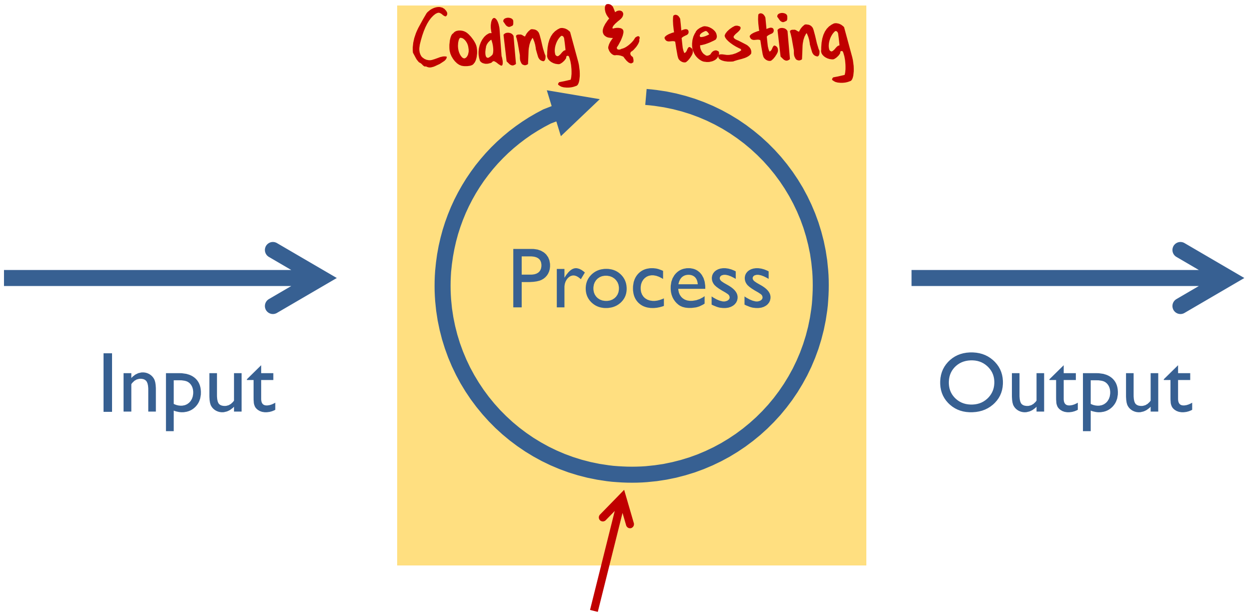
What's the ideal software development process?

- Build a shared mental model
  - Means a smaller spec
  - Less misunderstanding
- Have frequent feedback
  - Make sure you are going in the right direction
  - Course correction if goals change
- Value effectiveness over speed
  - No point going fast in the wrong direction

The software development process



The software development process



Coding & testing

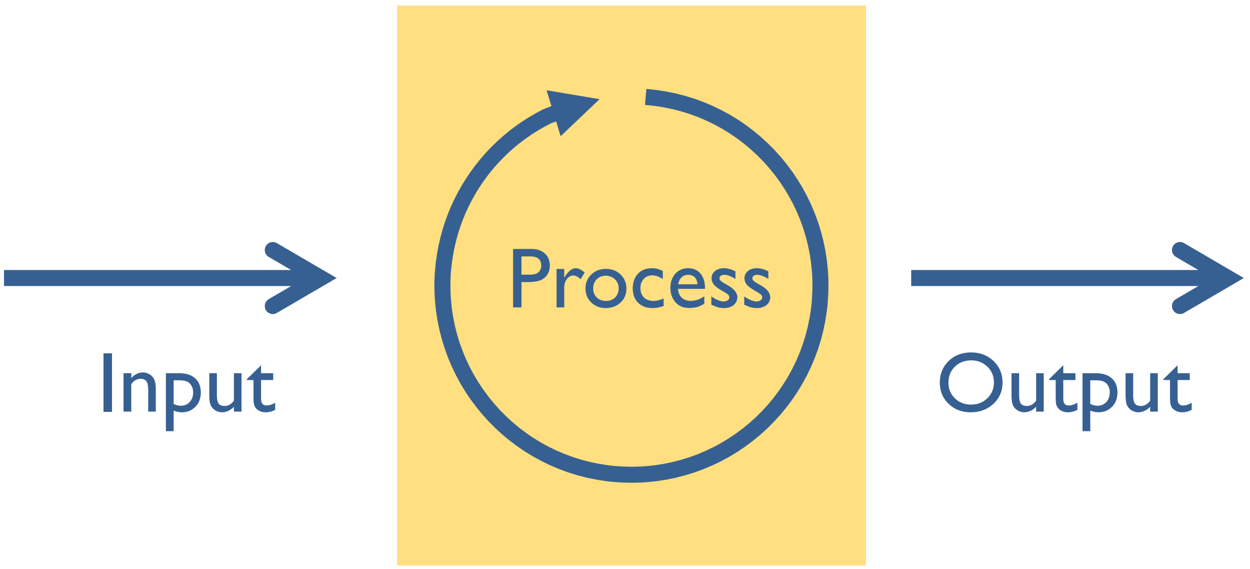
Input

Process

Output

Developers love to focus on this

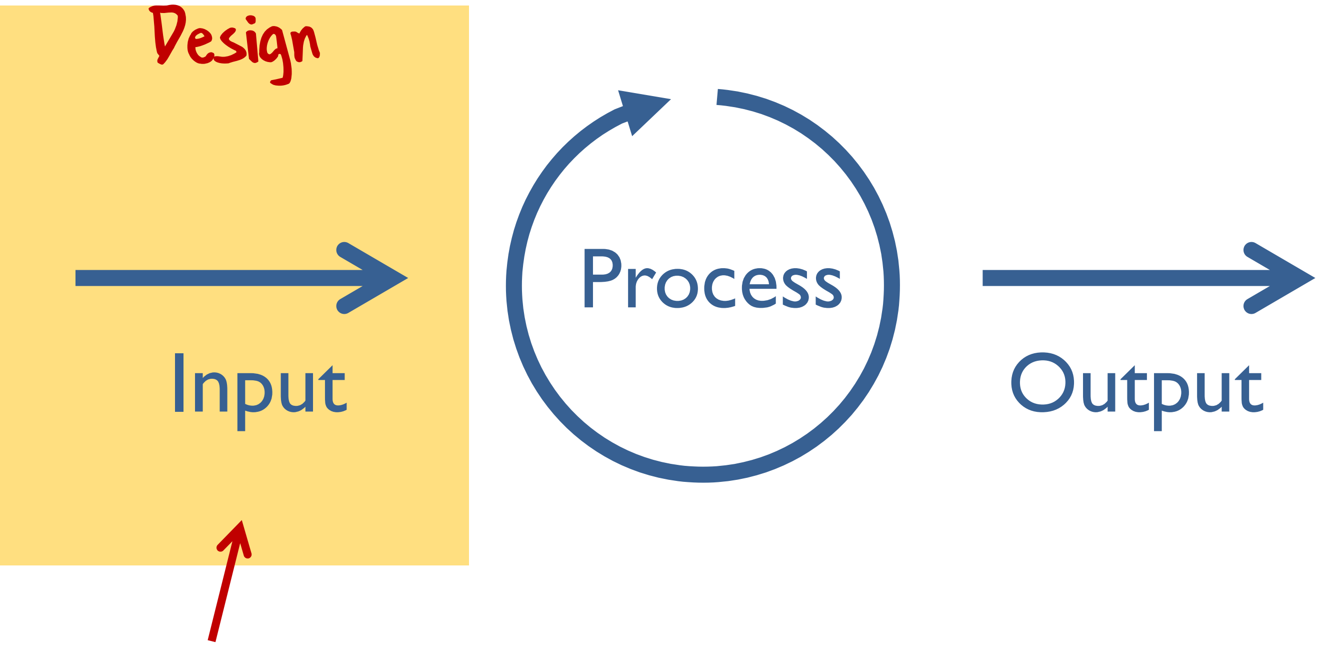
The software development process



Garbage in

Garbage out

The software development process



Design

Input

Process

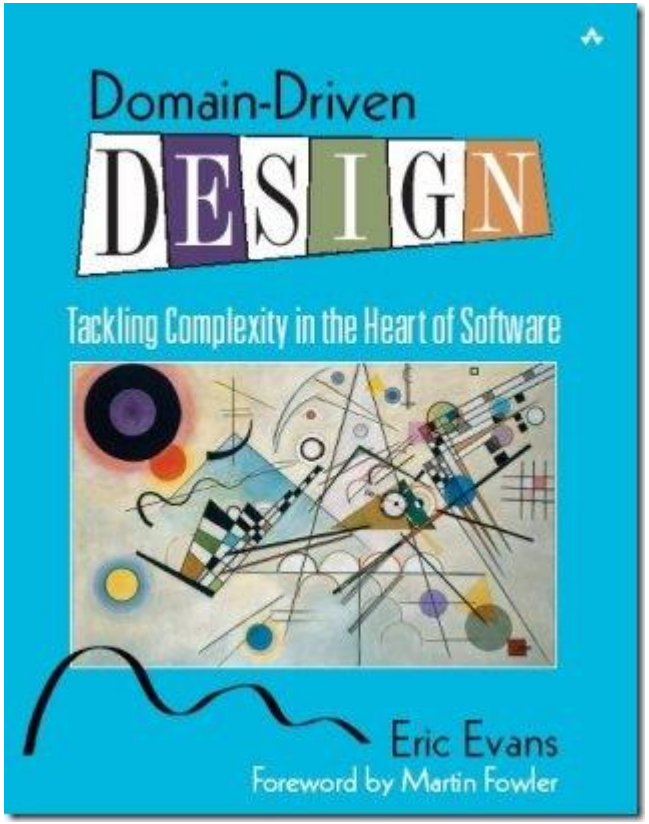
Output

This workshop

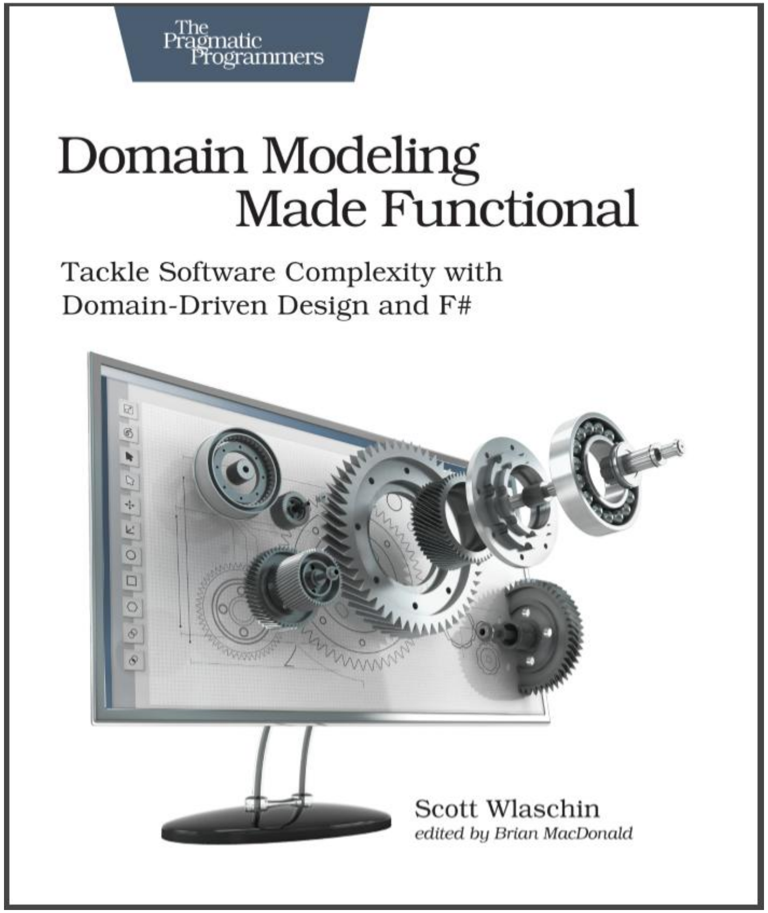
↓

(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#

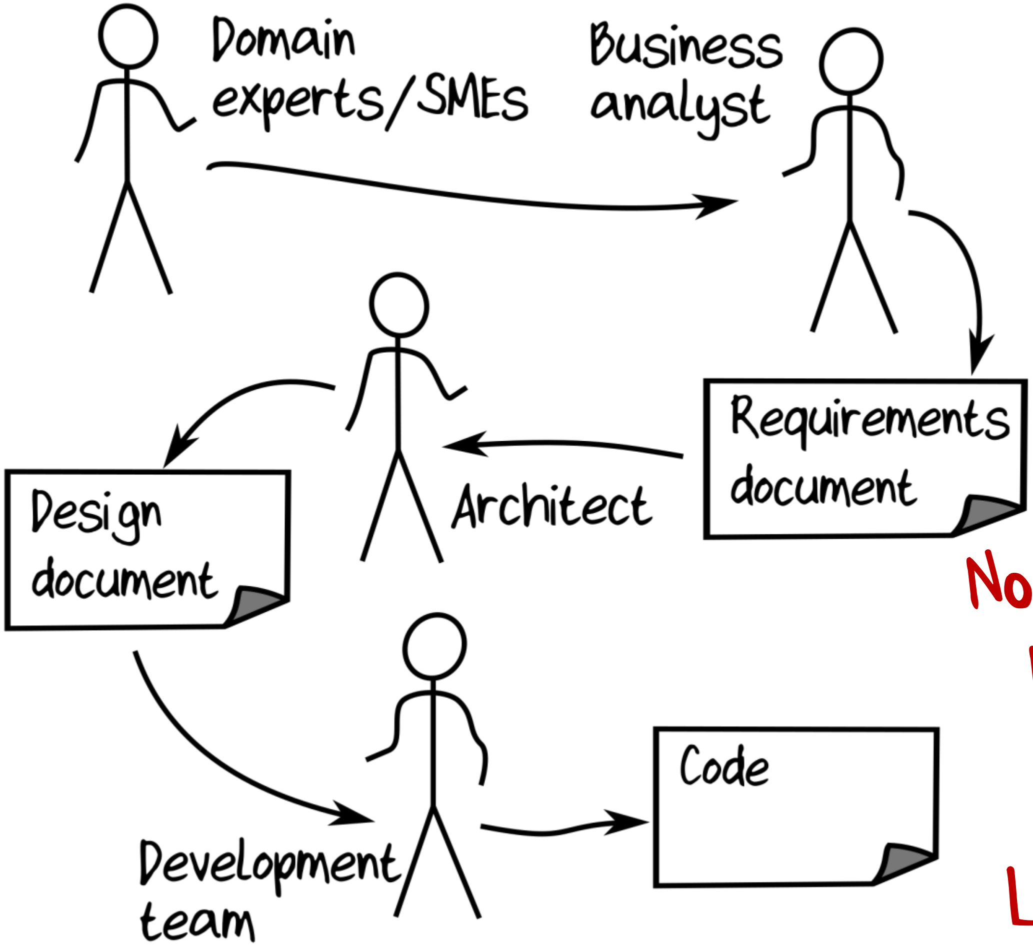
Scott Wlaschin

edited by Brian MacDonald

Or read the first 2 chapters of my book!

Why Domain-Driven Design?

Waterfall



Domain experts/SMEs

Business analyst

Requirements document

Architect

Design document

Development team

Code

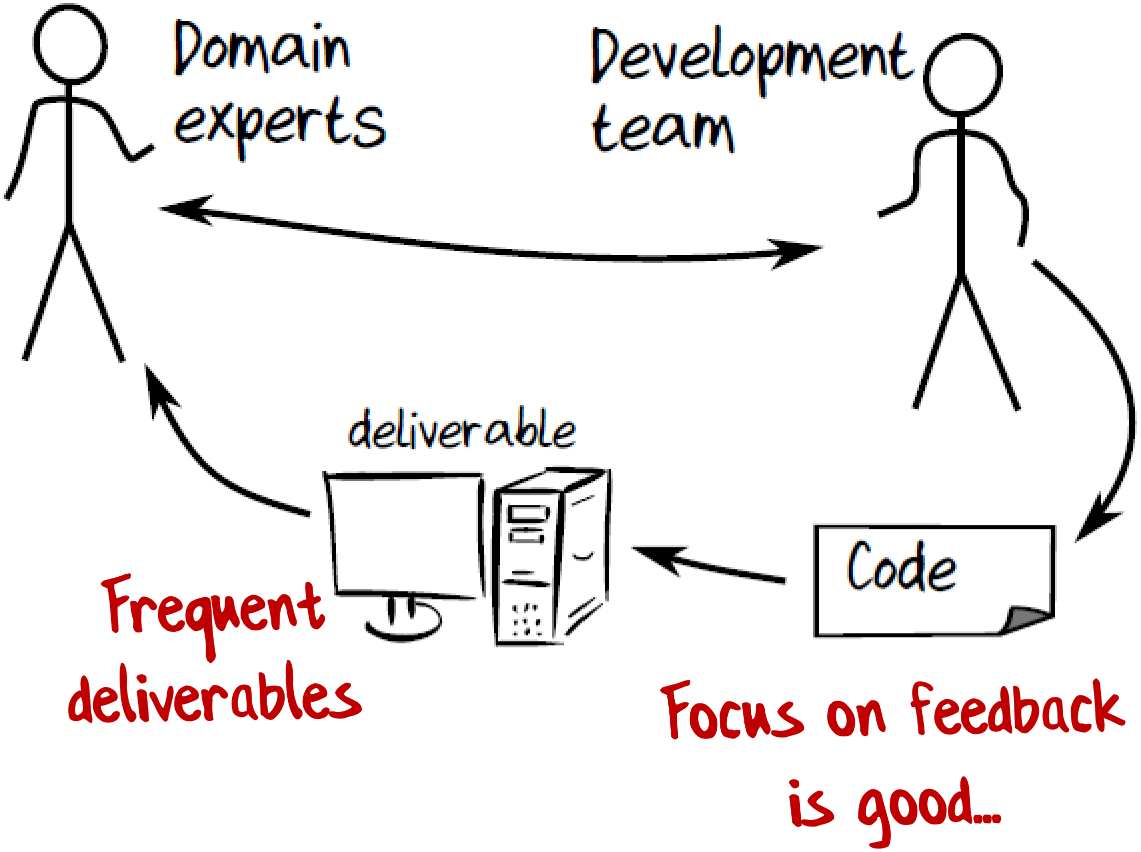
No shared model

No feedback

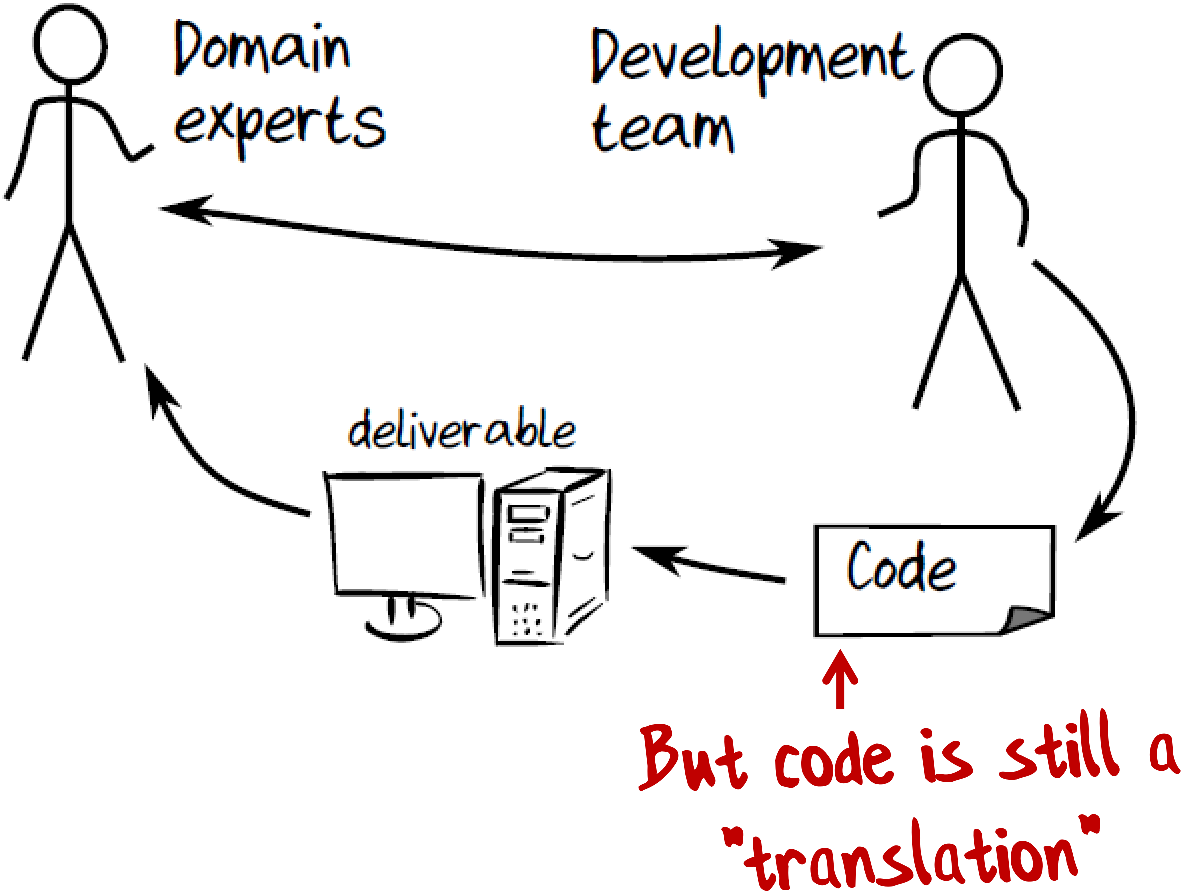
Like a bad children's game...

Warning: It's the developer's understanding of the domain that gets deployed!

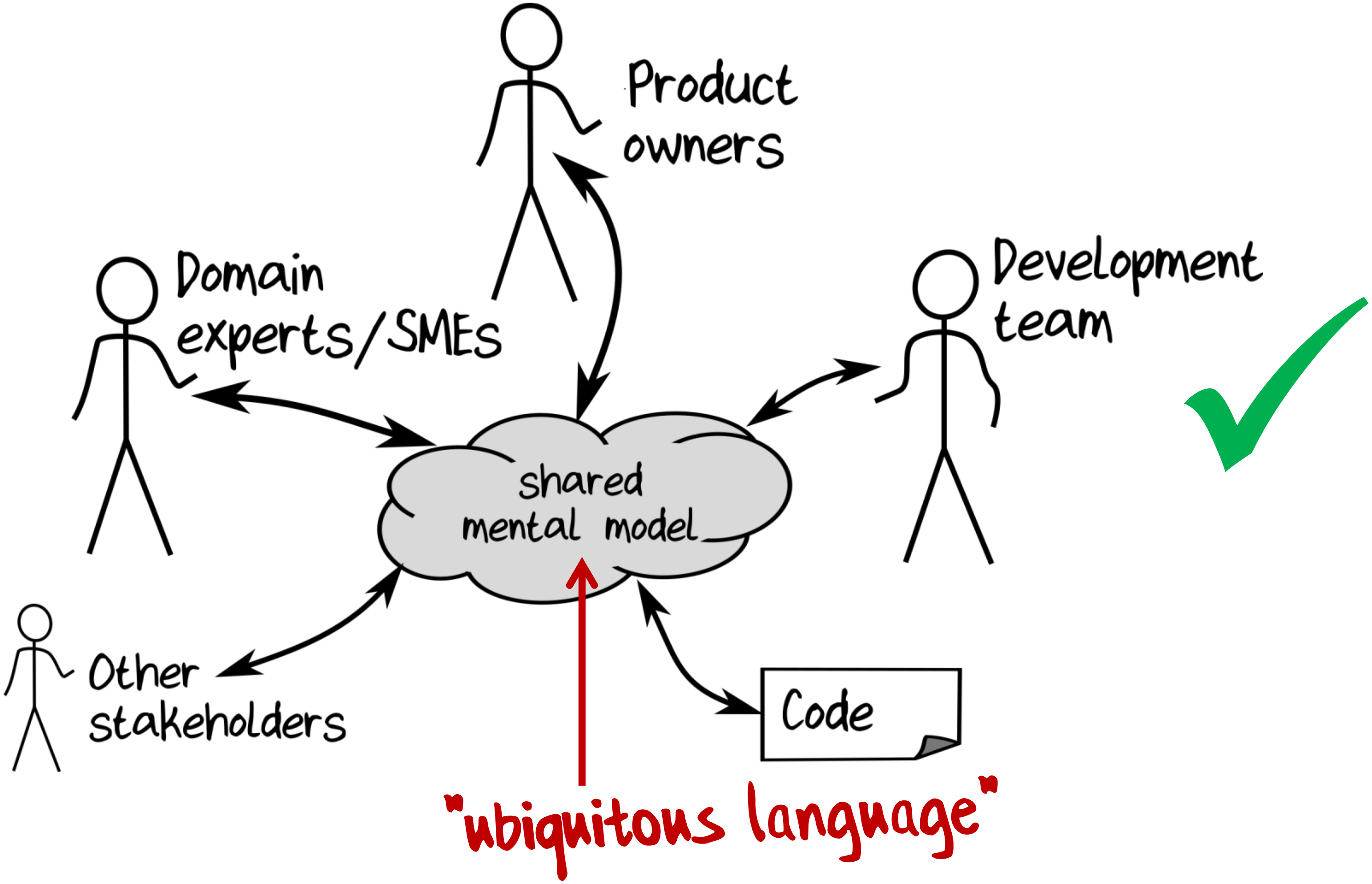
Agile

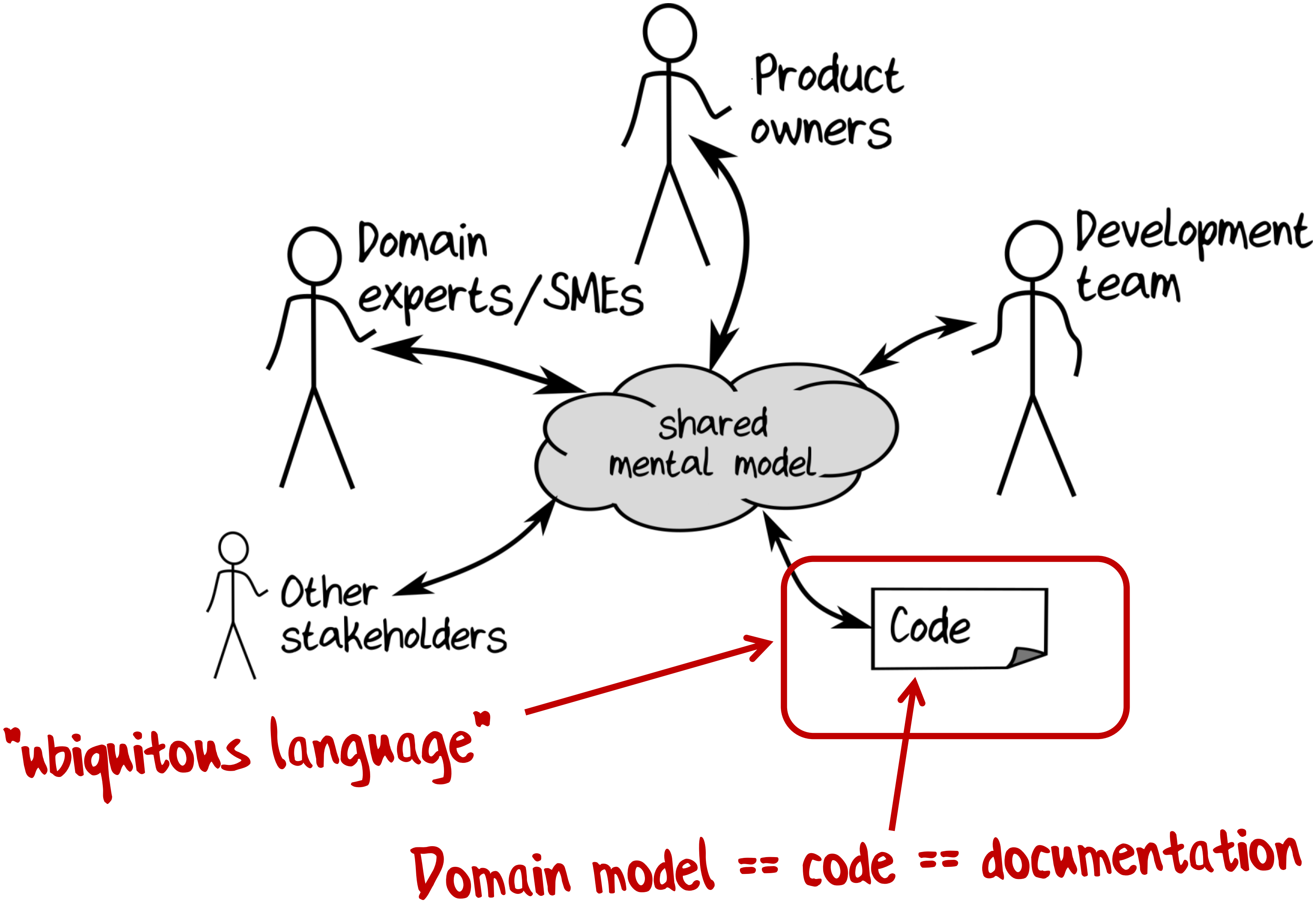

Domain experts
Development team
deliverable
Code
Frequent deliverables
Focus on feedback is good...

Agile


Domain experts
Development team
deliverable
Code
But code is still a "translation"

Domain-Driven Design





"ubiquitous language"

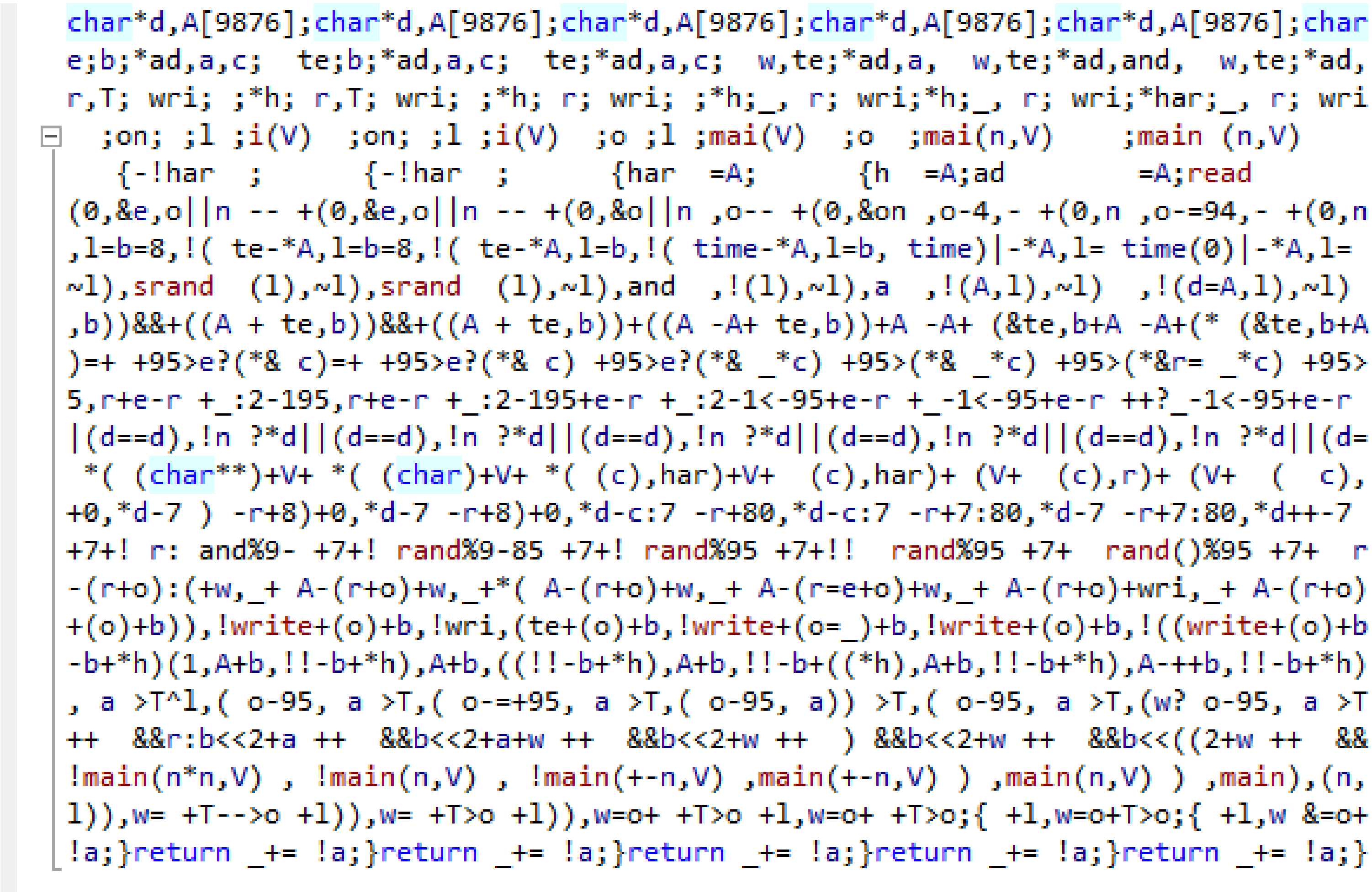
Domain model == code == documentation

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 some source code looks like



```
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

'|' 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

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

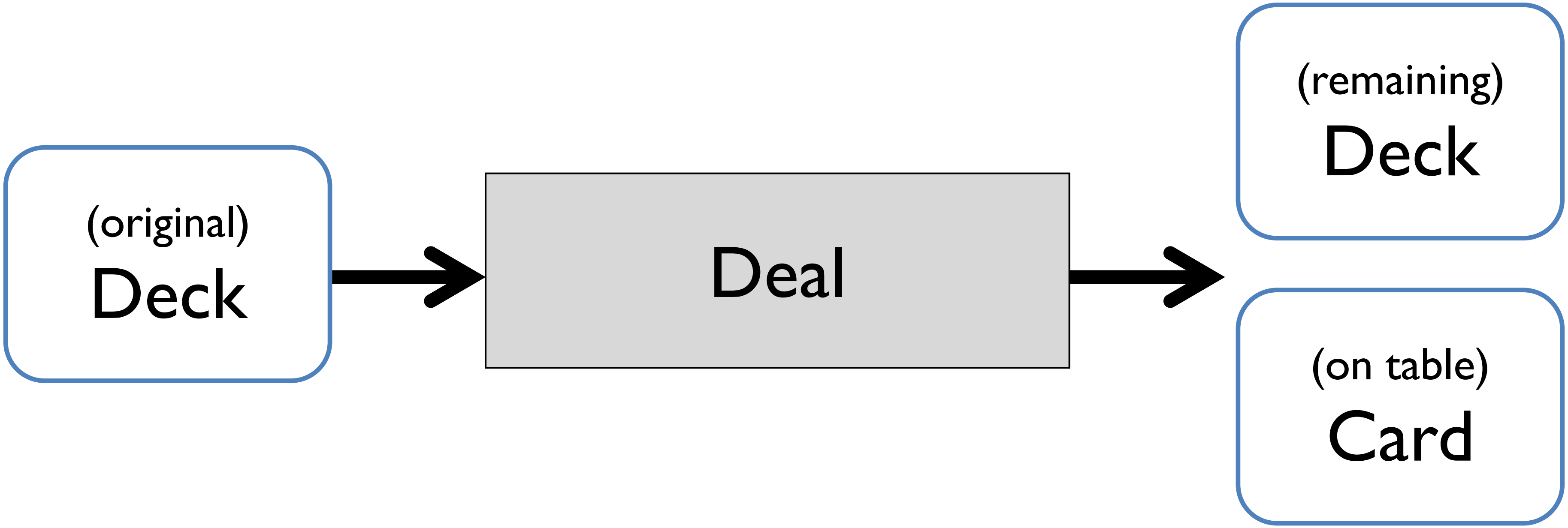
list type is built in

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

X -> Y means a workflow
- input of X
- output of Y

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

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

Modeling an action with a function

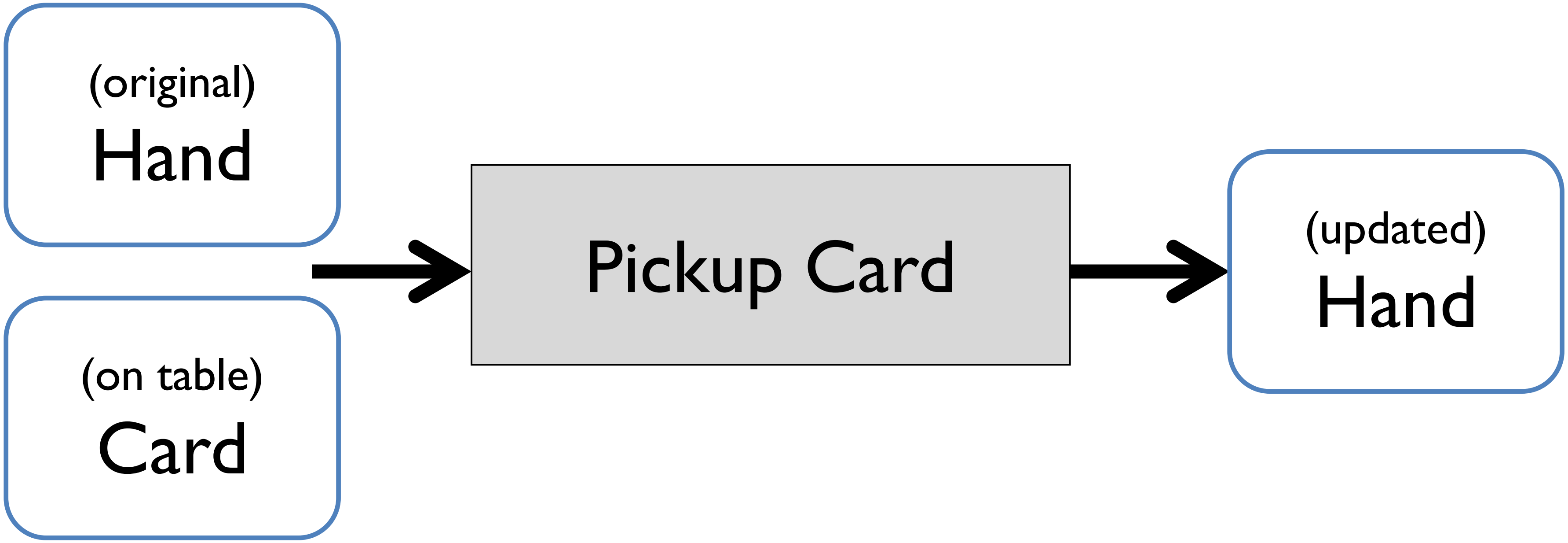


```
type Deal = Deck -> (Deck * Card)
```

Input

Output

Modeling an action with a function



(original)
Hand

(on table)
Card

Pickup Card

(updated)
Hand

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

Input

Output

```
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!

Creating the domain model is an interactive process

```
...
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"

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."

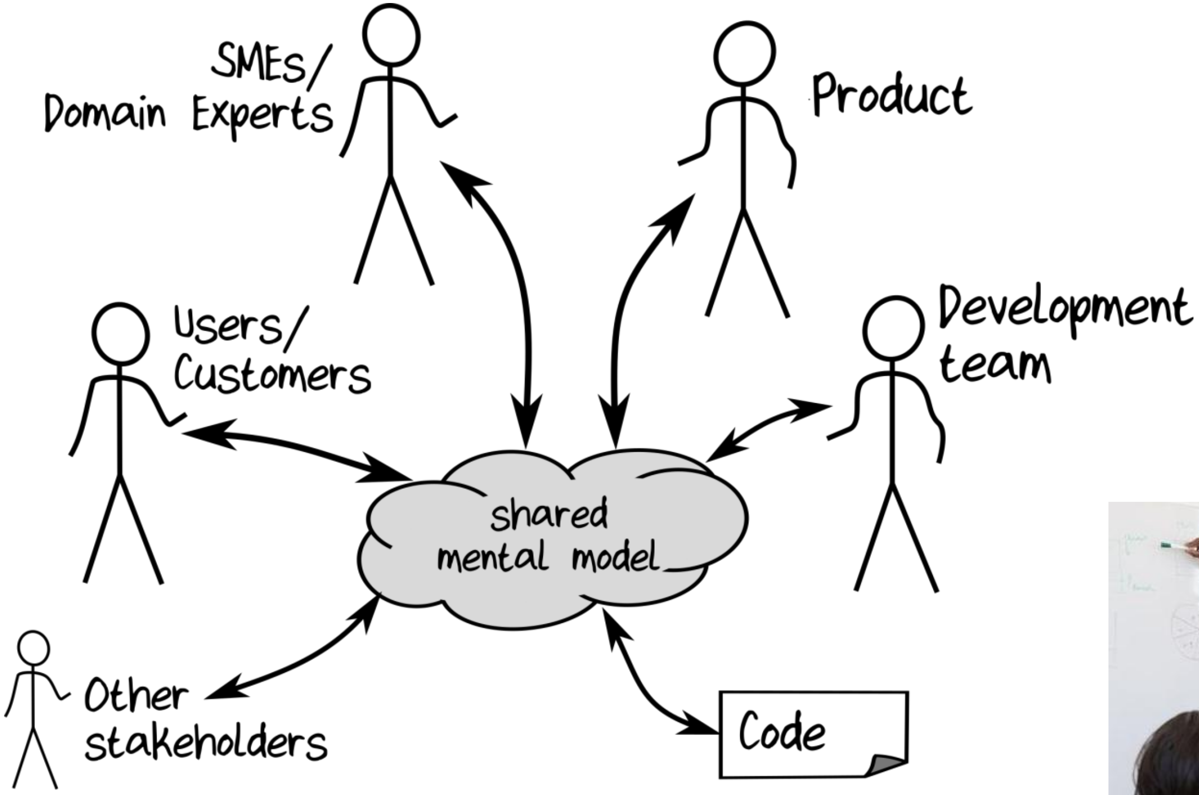
This is not pseudocode –
this is executable code!

```
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 just about the result.

The process of building the shared mental model is critical!

The process of building the shared mental model is critical!



Collaboration!

Conversation-driven development!

The schedule at a DDD conference...

Time	Talk
09:00	The art of talking to each other
10:00	You really should talk to each other more
11:00	Please, I beg you, stop focusing on tech and just talk to each other
12:00	Conversation-driven development
13:00	Our success story: we finally talked to each other
14:00	Report: Our project failed because we didn't talk to each other

Key DDD principle:

Communicate the design
in the code

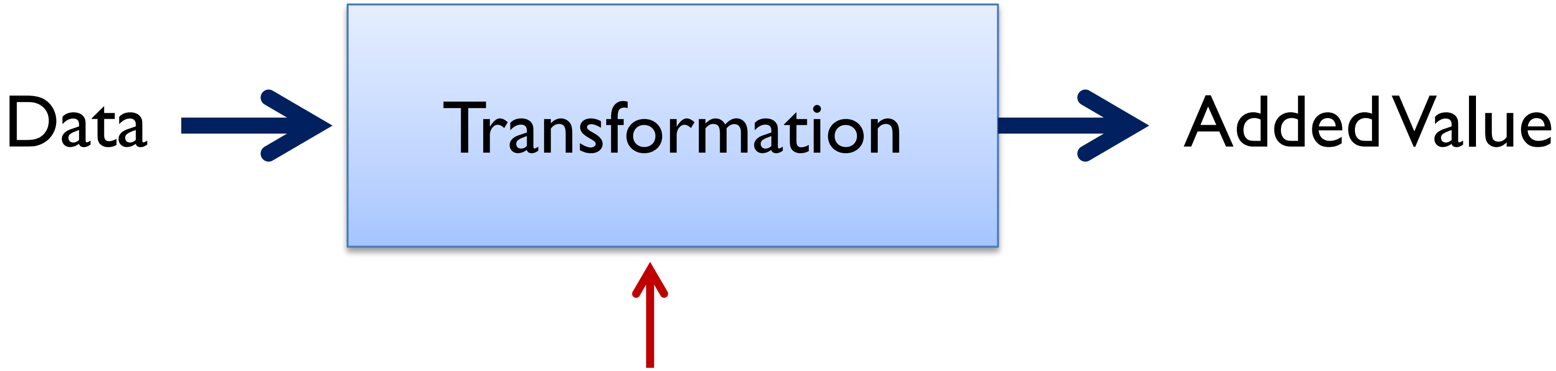
Part IV

The DDD approach to understanding the domain

Introducing "business events"

Data transformation

- A business doesn't just *have data, it transforms it somehow*
    - The value of the business is created in this process of transformation
- Data that is sitting there unused is not contributing anything



Data → Transformation → Added Value

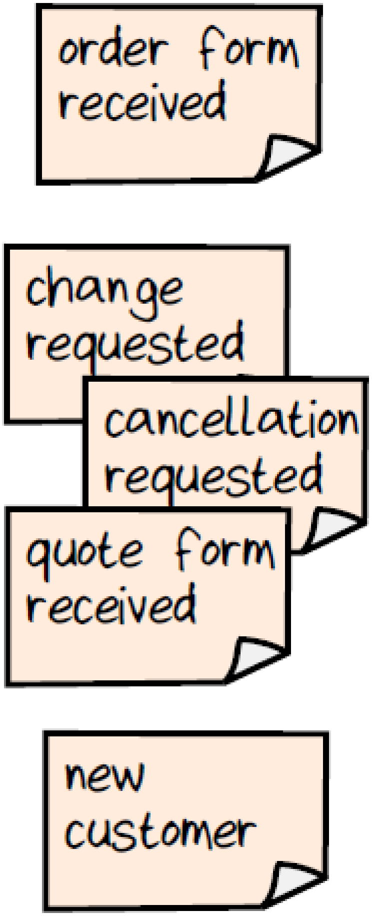
Categorization, added context, analysis, etc

P.S. This looks awfully like a function...

Events

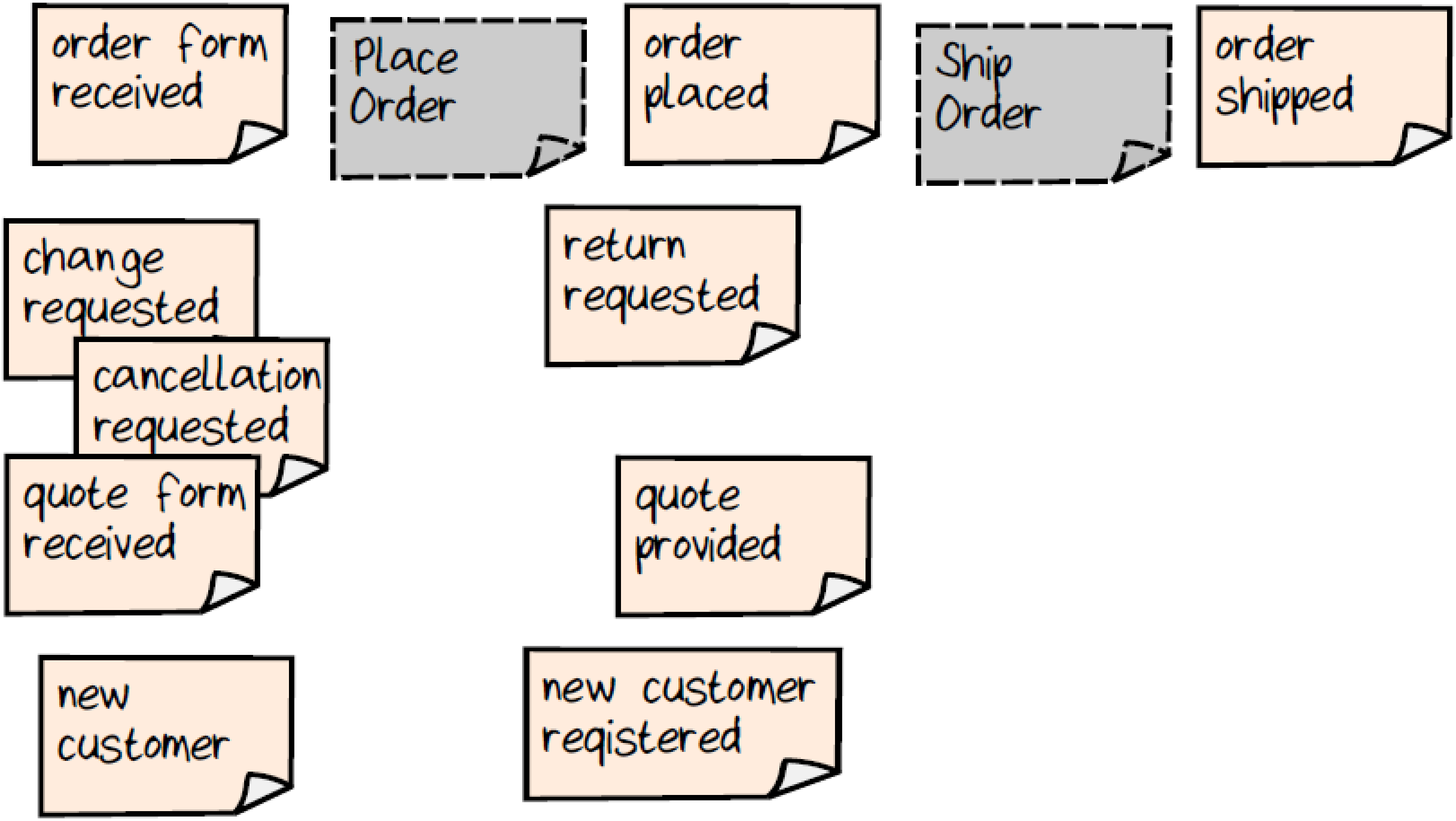
- What causes an employee or process to start working with that data and adding value?
  - An **event**!
- Examples of events:
  - New information arrives ("news")
  - Context changes
  - Customer asks for analysis

Finding the events with Event Storming

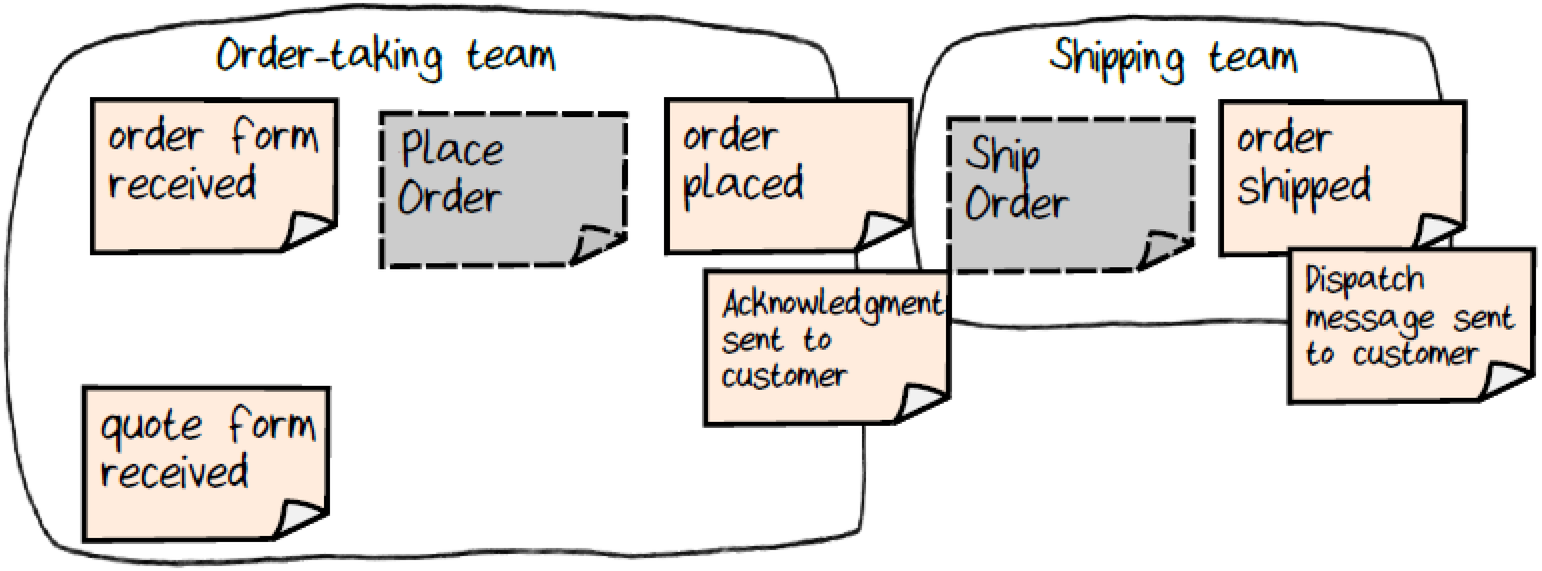


Order processing example

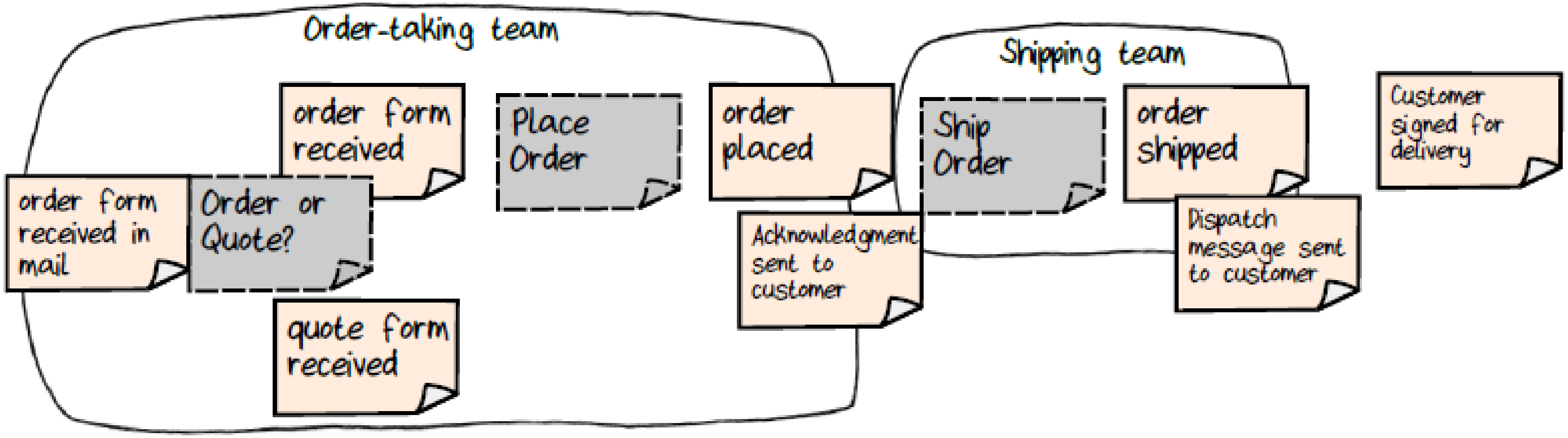
Finding the events with Event Storming



Then group the events

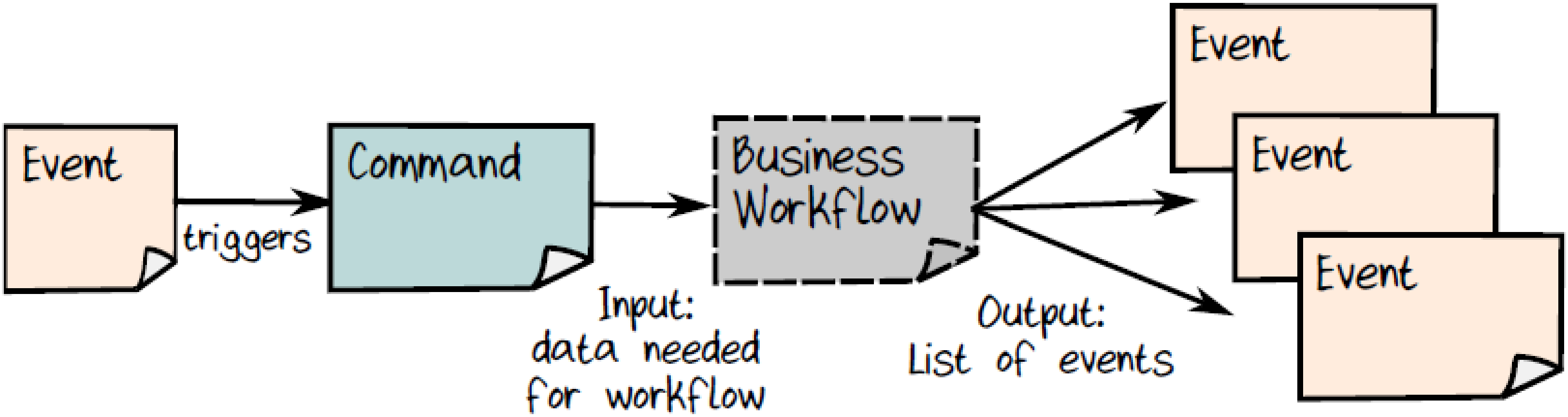


And extend to the edges



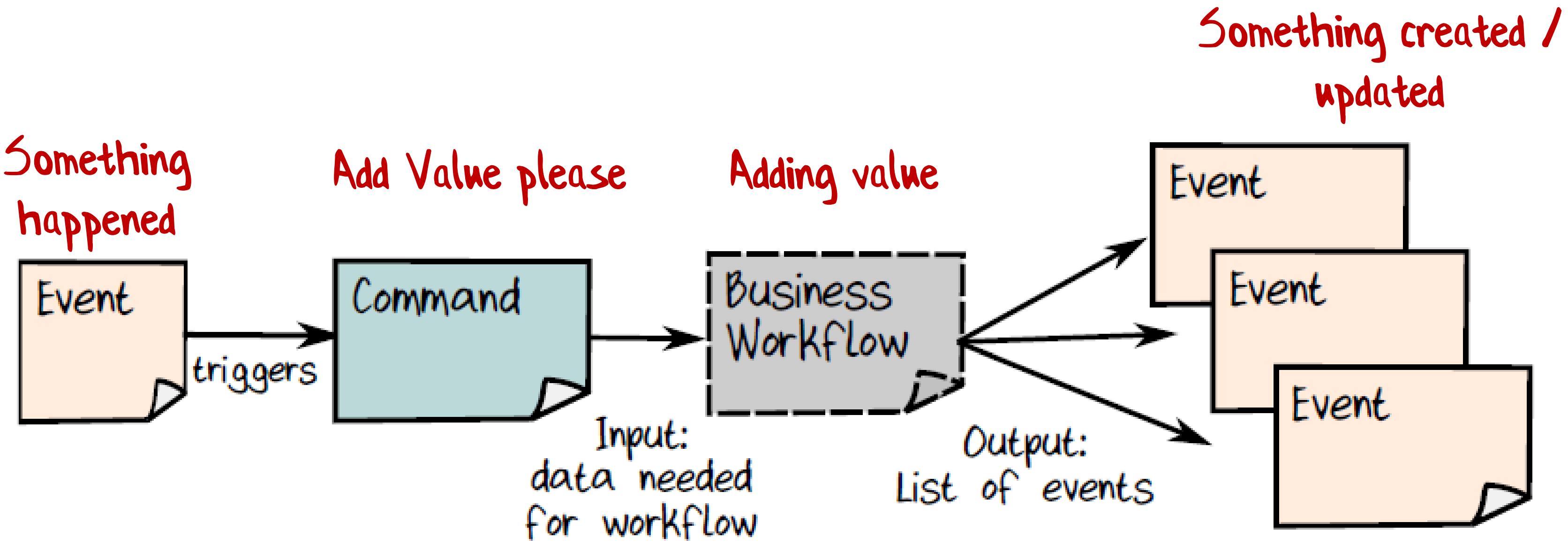
Introducing "workflows"

Events, commands, workflows



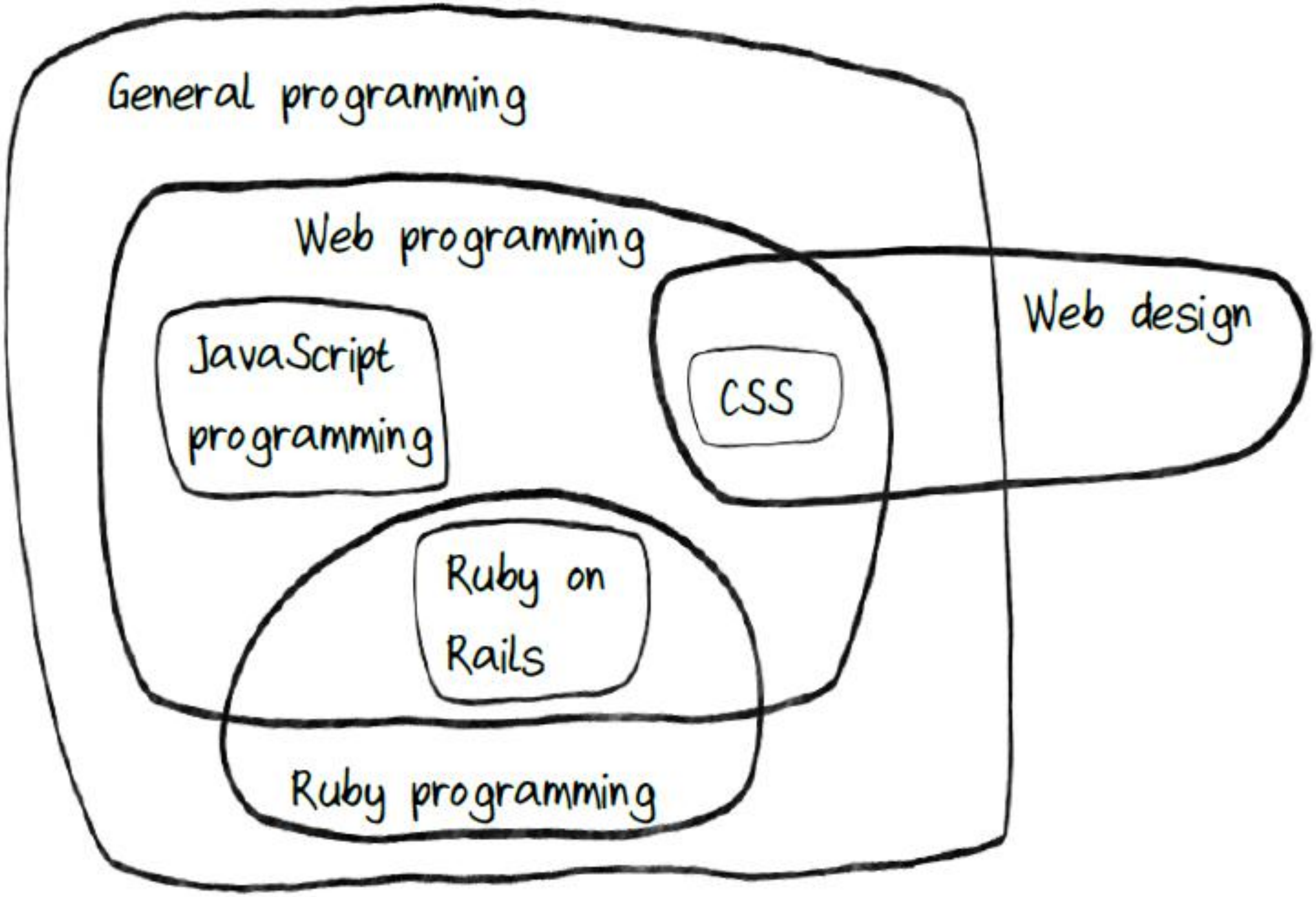
These are the units of work for specs, coding, & delivery

Events, commands, workflows

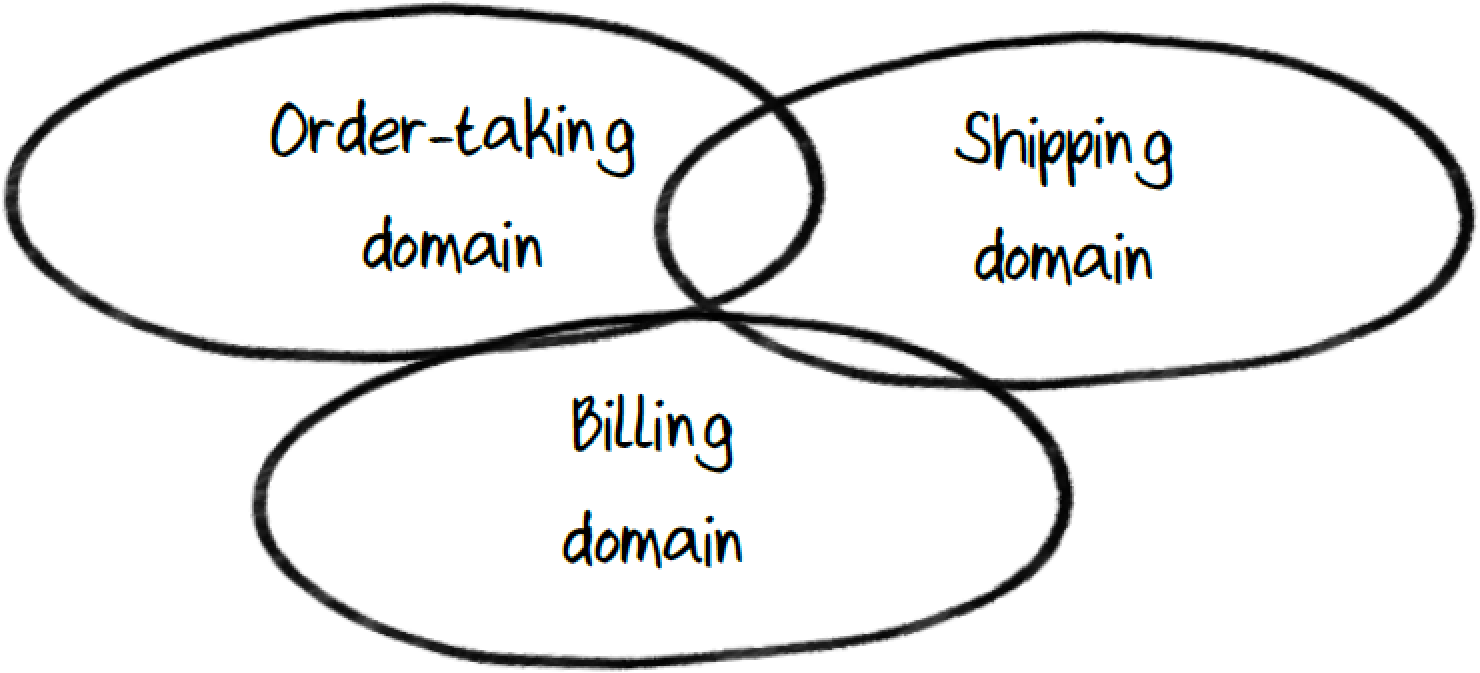


Introducing "subdomains"

What is a subdomain?



What is a subdomain?

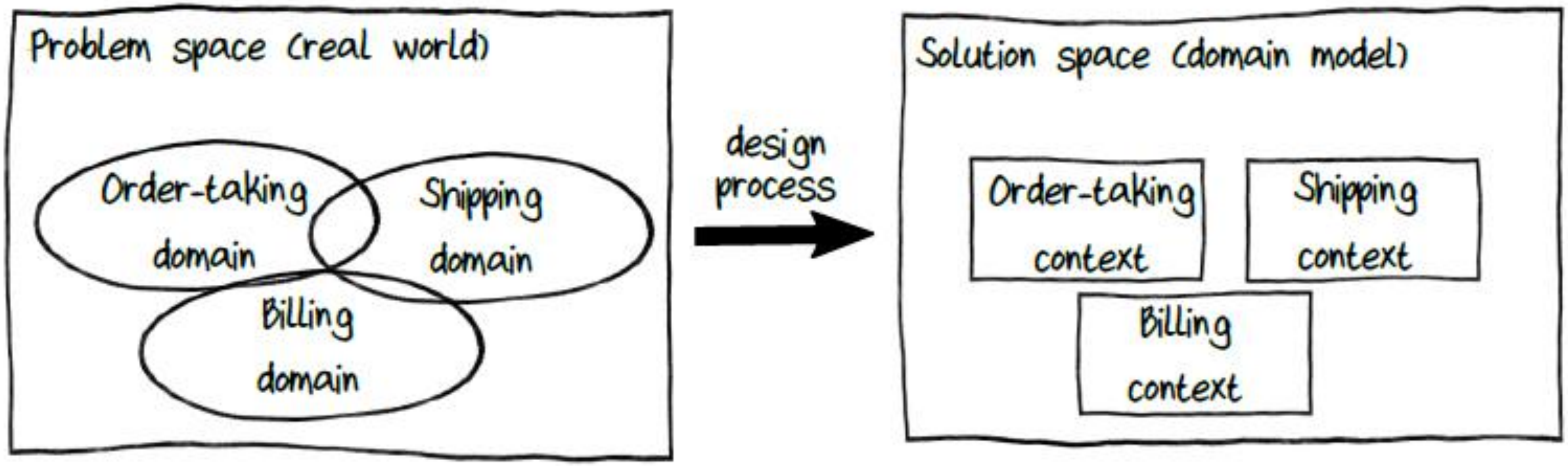


Introducing "bounded contexts"

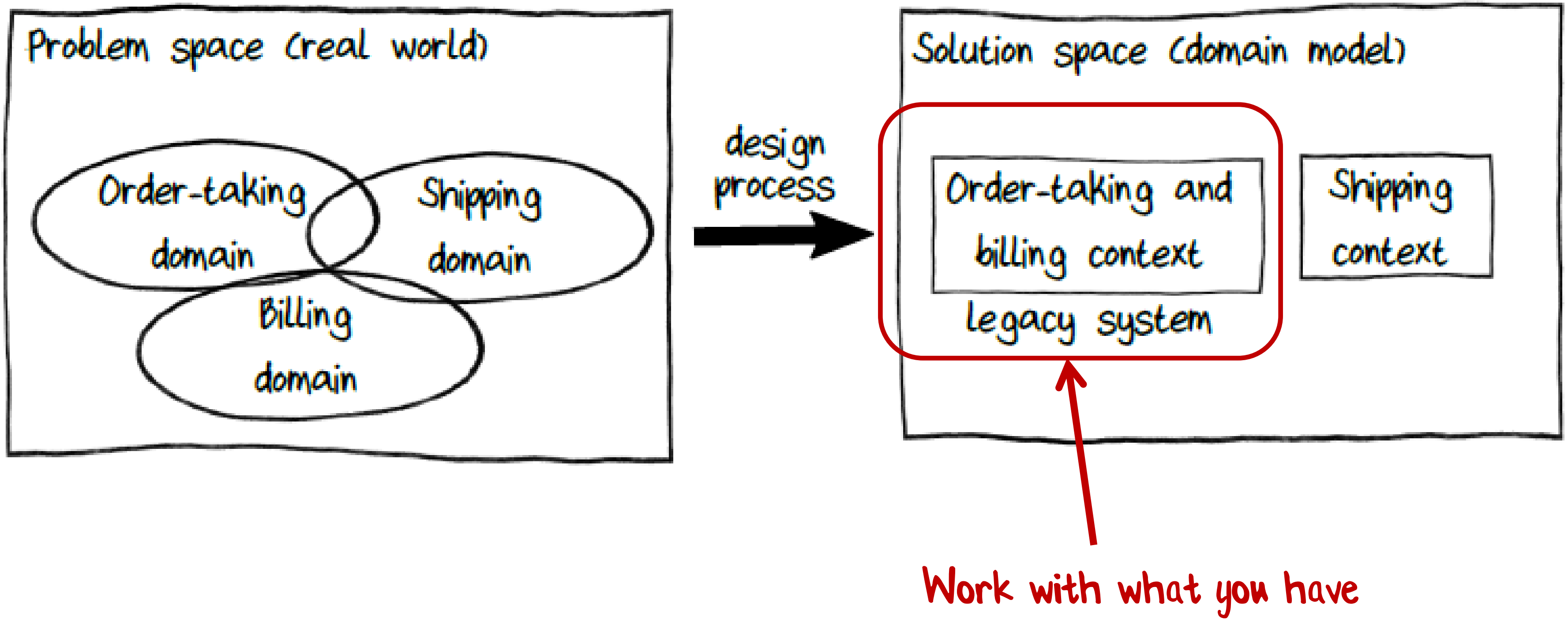
"Problem space" vs. "Solution space"

- The solution is a *model* of the problem domain
  - Only contains aspects of the domain that are relevant!
- A "Subdomain" is in the problem space
- A "Bounded context" is in the solution space

"Problem space" vs. "Solution space"



"Problem space" vs. "Solution space"



Why "Bounded Context"?

- Focus on what is important
  - being aware of the context
  - being aware of the boundaries.
- "*Context*"
  - Specialized knowledge and common language
  - Information taken out of context is confusing or unusable
- "*Bounded*"
  - We want to reduce coupling
  - Contexts must evolve independently!

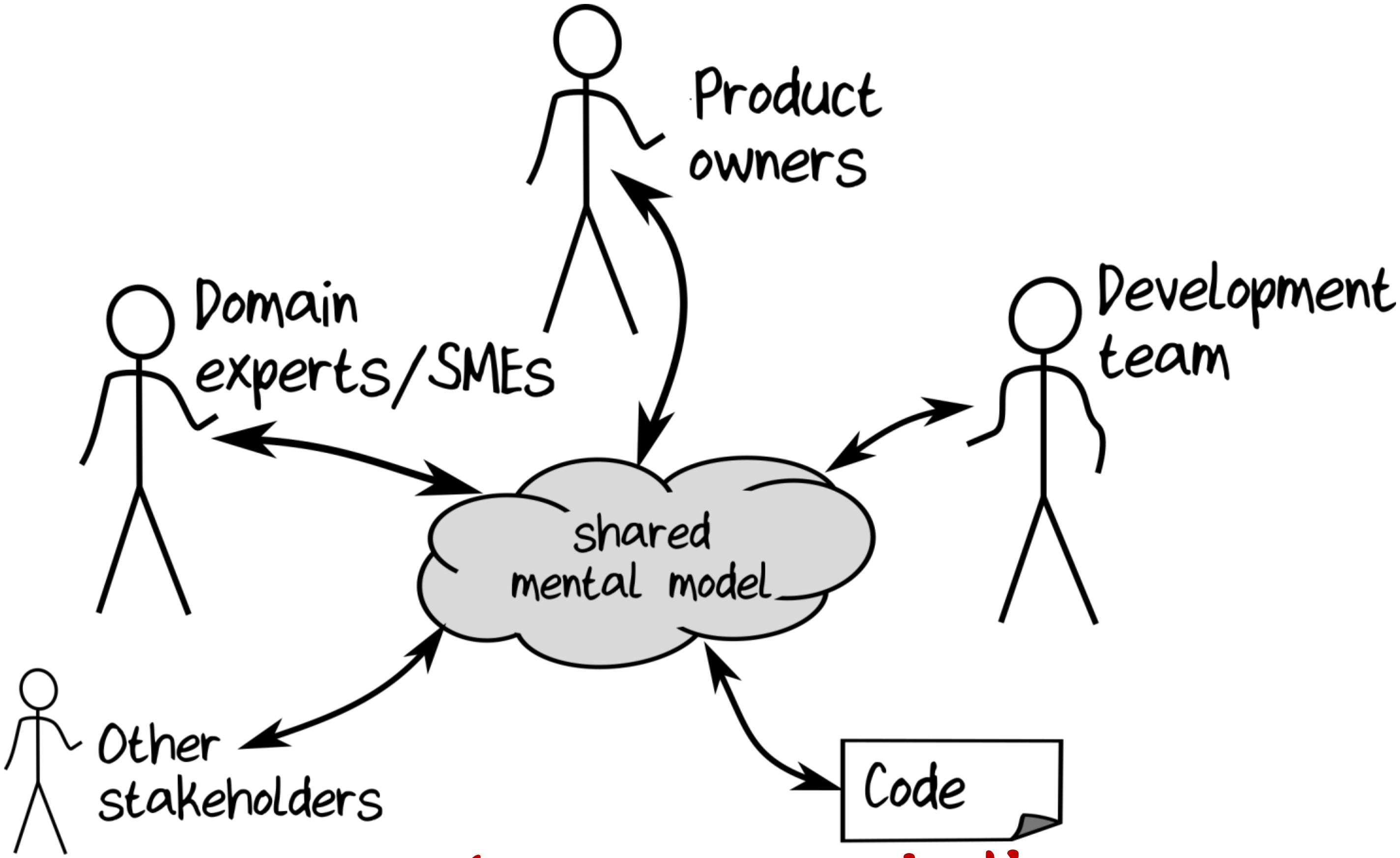
How to get the contexts right

- Listen to the domain experts!
  - Pay attention to existing team and department boundaries
- Don’t forget the “bounded” part of a bounded context
  - Watch out for scope creep when setting boundaries
- Design for autonomy
  - If two groups contribute to the same bounded context, they might end up pulling the design in different directions as it evolves (a three legged race!)
  - Better to have separate bounded contexts that can evolve independently than one mega-context that tries to make everyone happy

Introducing "ubiquitous language"

Ubiquitous Language

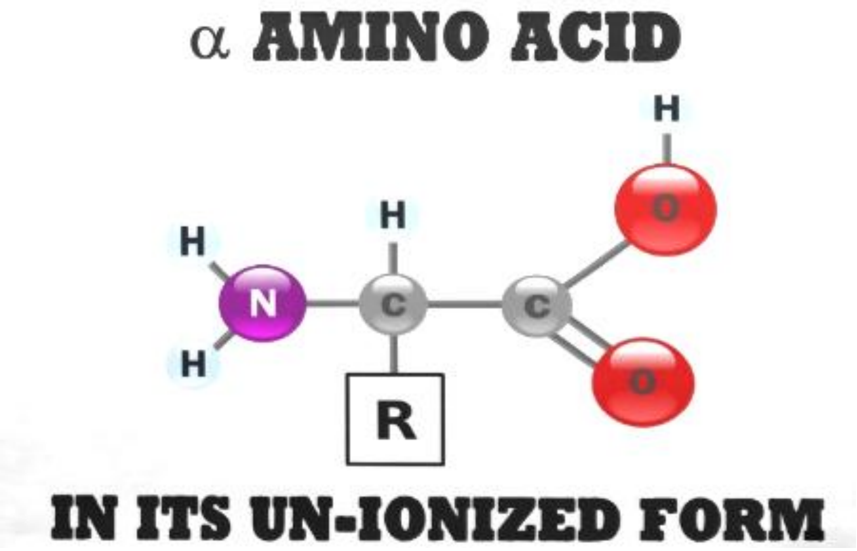
- The Ubiquitous Language is a set of concepts and vocabulary associated with the domain and is shared by
  - Domain experts
  - Product owners
  - Development team
  - The source code
- The "everywhere language"

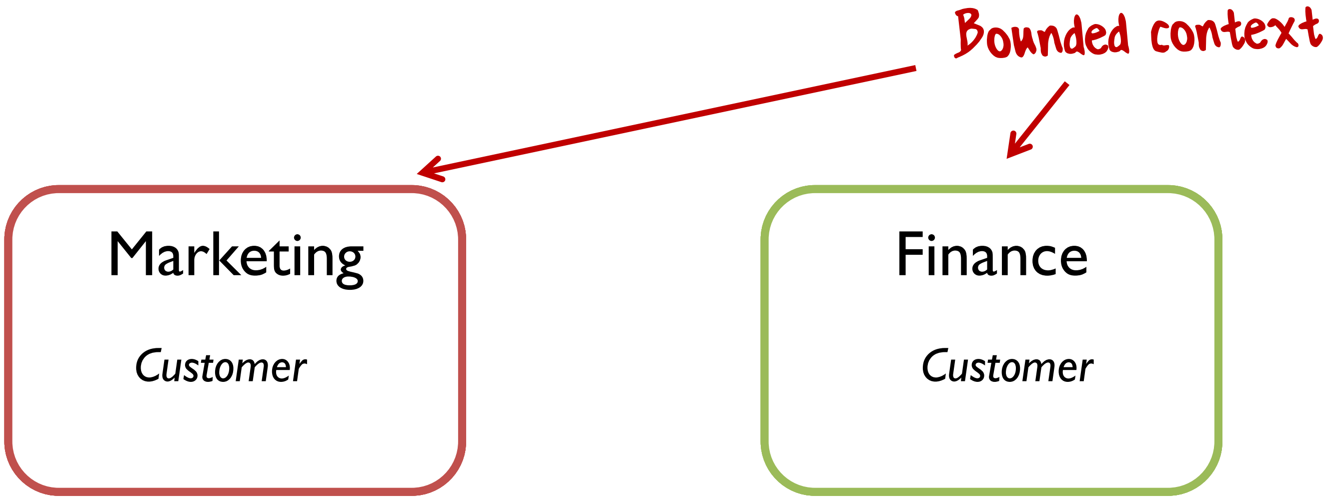


Shared communication can be hard!

U-N-I-O-N-I-Z-E

NIÑOS UNIDOS
BETTER
MEDICAL
SCHOOLS
GIFT PACKAGES
BETTER
MEDICAL
SCHOOLS


α AMINO ACID
H
H
H
N
C
C
O
H
O
R
H
IN ITS UN-IONIZED FORM



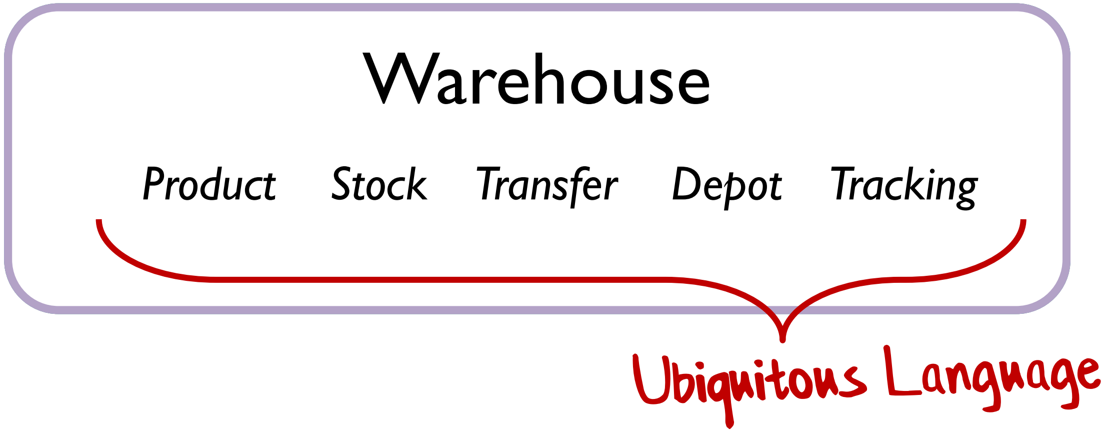
Bounded context

Marketing

Customer

Finance

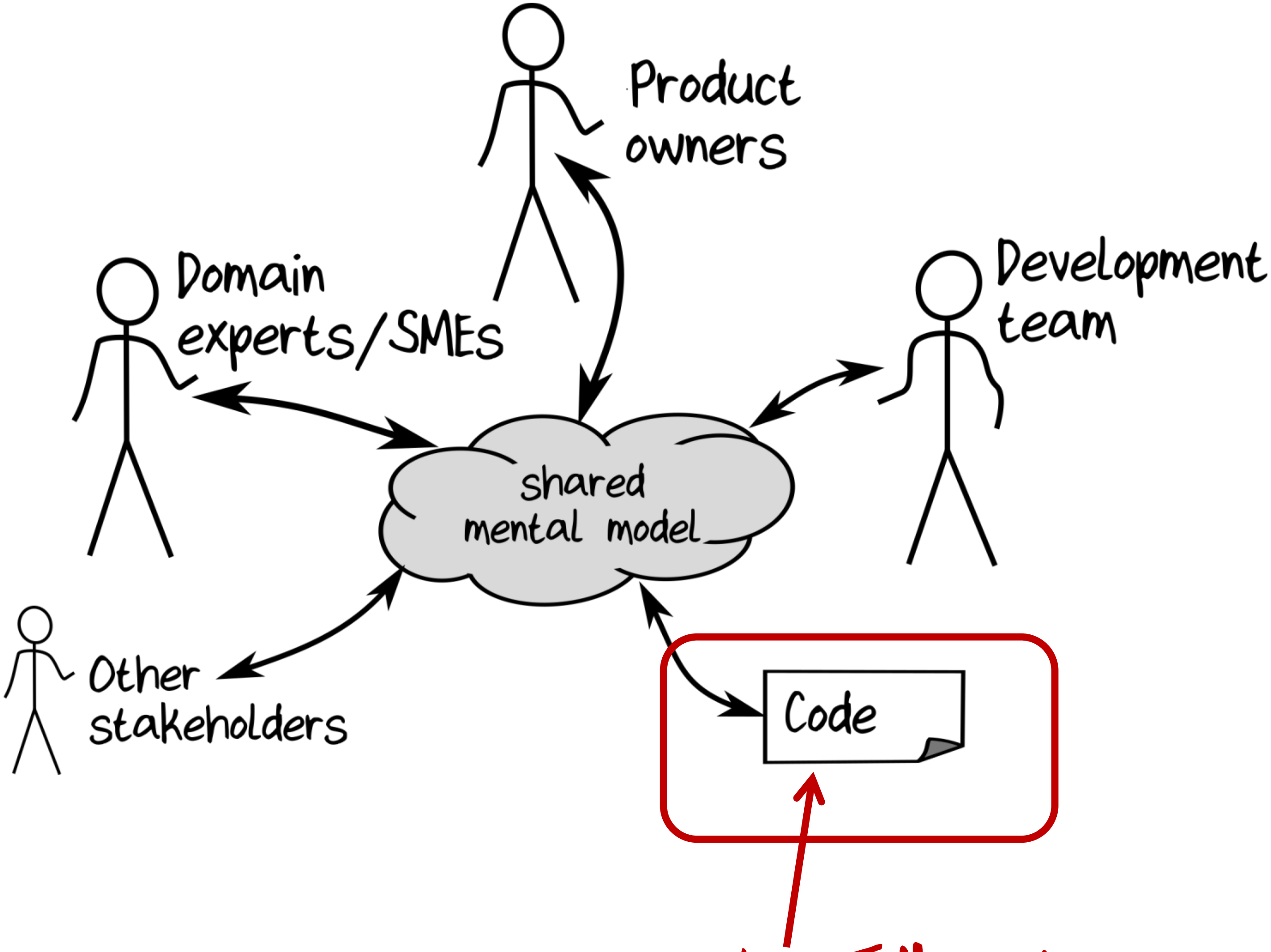
Customer



Warehouse

Product *Stock* *Transfer* *Depot* *Tracking*

Ubiquitous Language



Product owners

Domain experts/SMEs

Development team

shared mental model

Other stakeholders

Code

Domain model in F# code

Bounded context

```
module CardGame =
```

Ubiquitous language

```
    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}
```

Workflow

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

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

Summary of DDD concepts

- Domain Model
  - A set of **simplifications** that represent those aspects of a domain that are **relevant** to a particular problem.
  - The domain model is part of the **solution space**
- Bounded context
  - A subsystem in the domain model
  - Is autonomous and has explicit boundaries.

Summary of DDD concepts

- Ubiquitous Language
  - Concepts and vocabulary associated with the domain
  - Shared by both the team members and the source code.

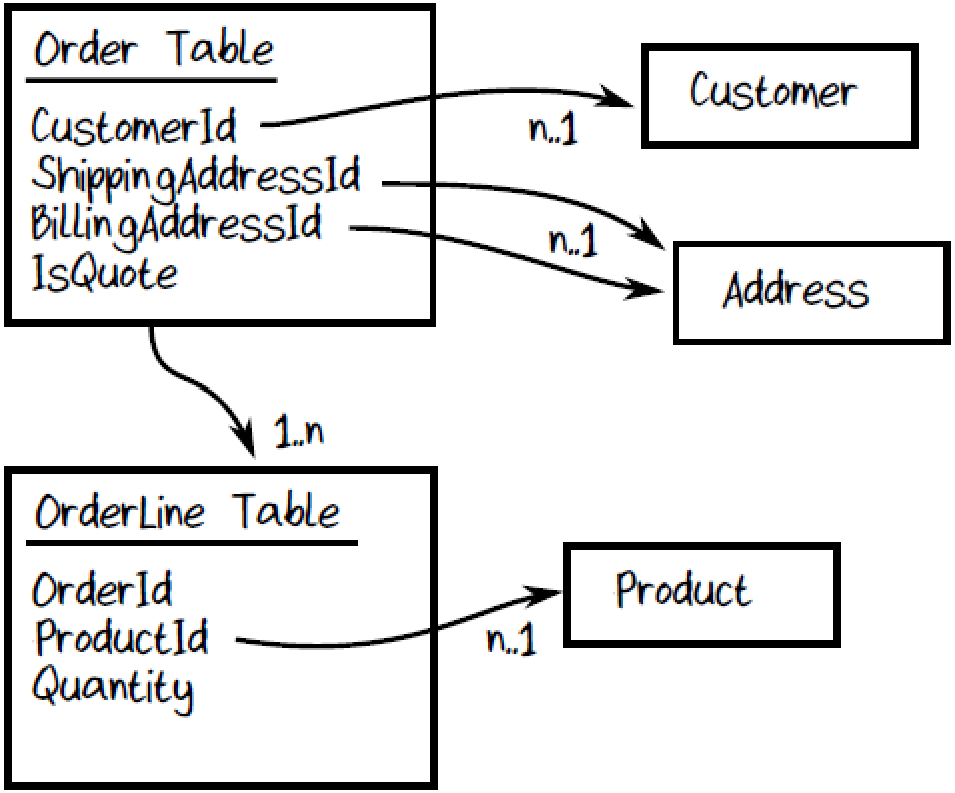
Summary of DDD concepts

- Domain Event
  - A record of something that happened in the system.
  - An event often triggers additional activity.
- Command
  - A request for some process to happen
  - Triggered by a person or another event.
  - If the command succeeds, the state of the system changes and one or more Domain Events are recorded.

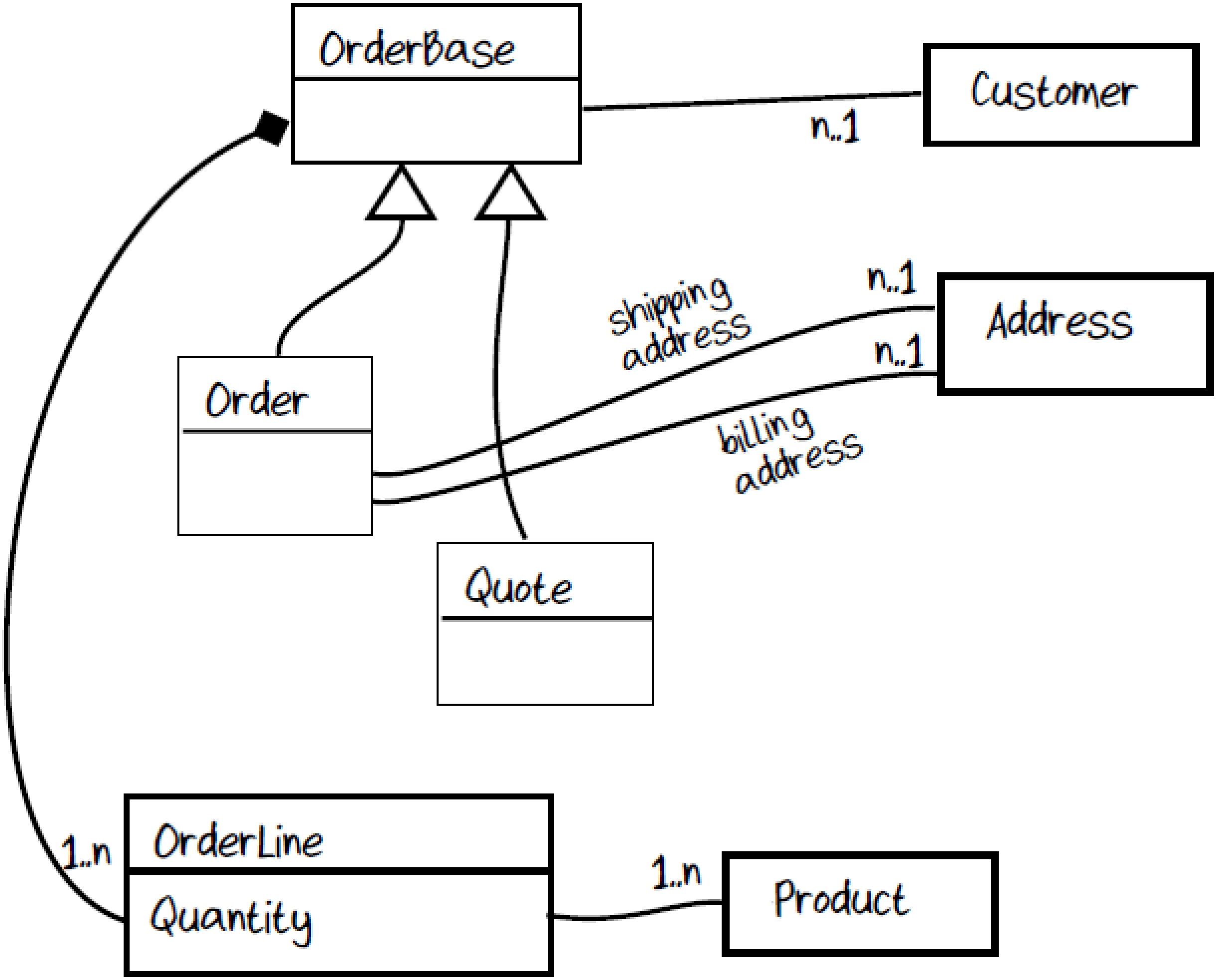
Part V

Getting started with domain modelling

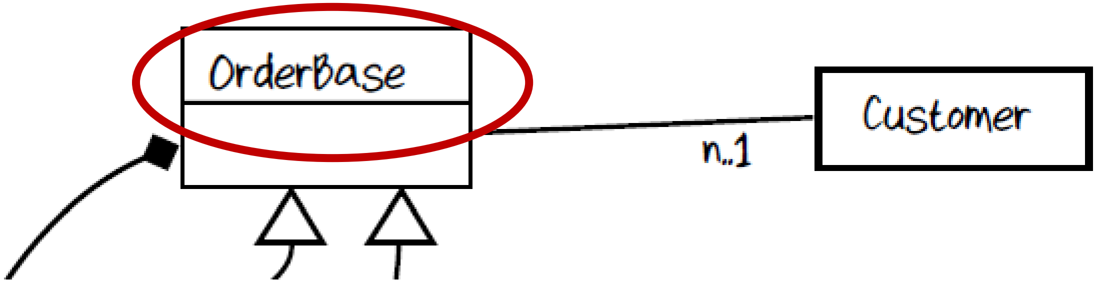
It's not database modelling!



It's not OO modelling!



It's not OO modelling!

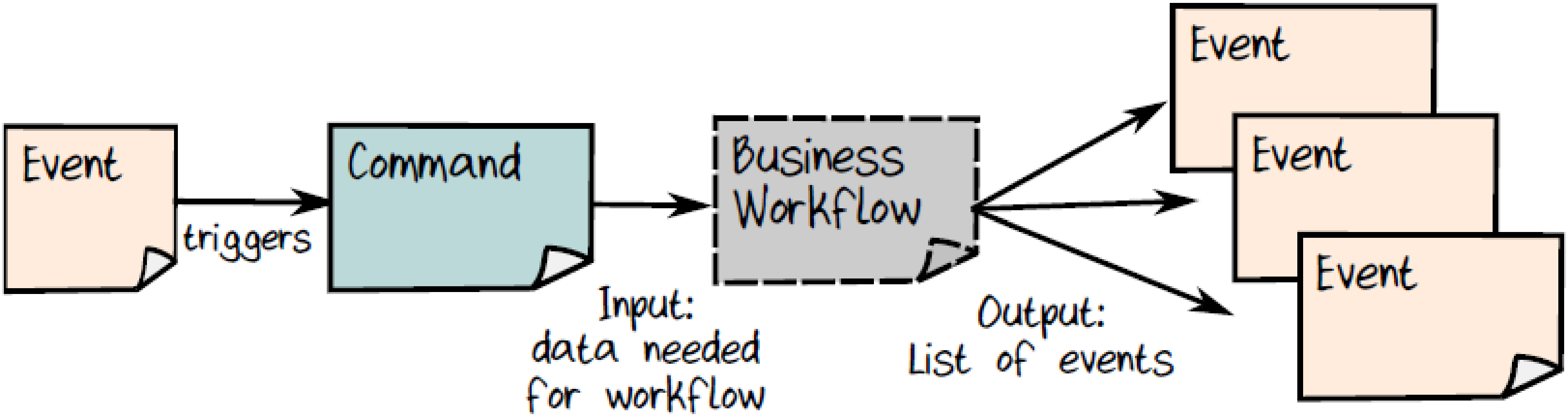


Pro tip: If you have a "base", "factory", "manager", or "helper" class then you're doing it wrong!

A domain expert or SME wouldn’t know what you meant by these words.

My recommended way of domain modelling

Events, commands, workflows



These are the units of work for specs, coding, & delivery

Domain modelling guidelines

- Start with the events
- What data is needed?
  - From the event itself
  - From the current state of the system
  - What is optional vs. required?
- What are the output events?
  - For broadcast to downsteam systems

Domain modelling guidelines

- What is the change in the system state?
  - Is this a state transition?
- What are the side effects?
  - Things that must be done but are internal to the domain
- What can go wrong?
  - Model success or failure with an OR choice

Domain modelling guidelines

- If you hear "it depends", drill deeper
- Disagreement between experts is OK
  - The design process is about getting *everyone* on the same page.
- Everyone must be in the room
  - Otherwise they miss being part of the process. It's not just about the results of the process

Start with an event and workflow

```
Bounded context: Order-Taking

Workflow: "Place order"
triggered by:
  "Order form received" event
primary input:
  An order form
other input/dependencies:
  Product catalog to lookup prices
output events:
  "Order Placed" event
side-effects:
  An acknowledgment is sent to the customer,
  along with the placed order
```

Document the data with AND

```
data Order =
  CustomerInfo
  AND ShippingAddress
  AND BillingAddress
  AND list of OrderLines
  AND AmountToBill

data OrderLine =
  Product
  AND Quantity
  AND Price

data CustomerInfo = ??? // don't know yet
data BillingAddress = ??? // don't know yet
```

Never use primitive types in a domain model

```
data Customer = string AND string
data OrderLine = int AND int
```

✗

Important! "int" and "float" are not domain concepts

```
data CustomerName = FirstName AND LastName
data OrderLine = ProductId AND Quantity
```

✓

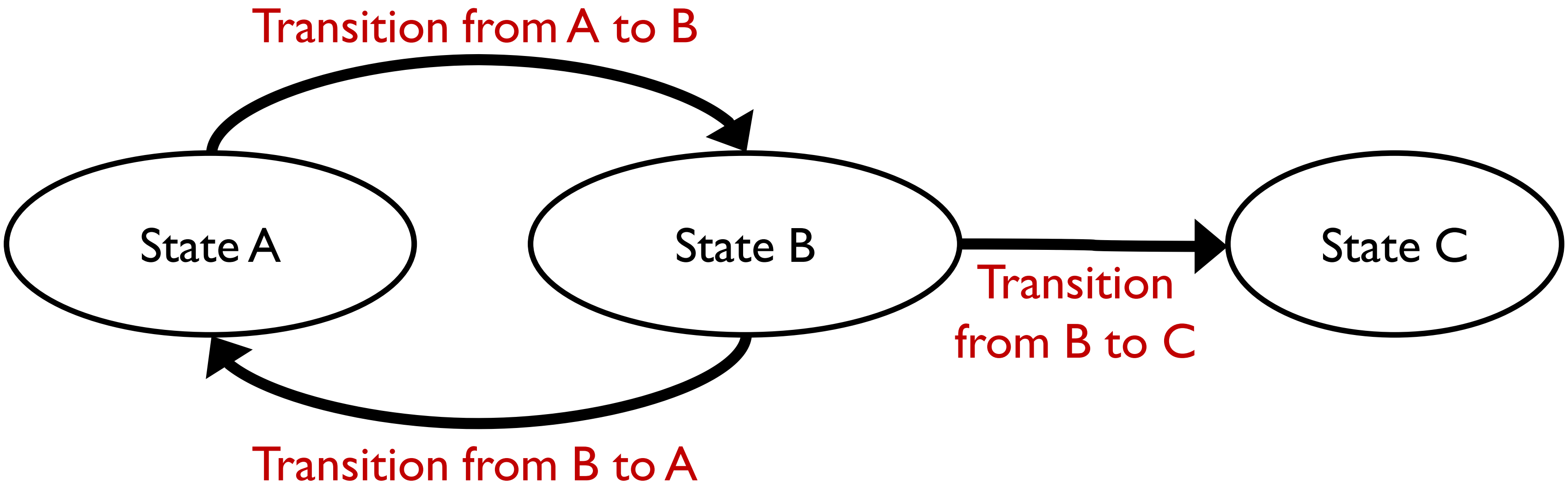
Document choices with OR

```
data ContactInfo =
  EmailAddress
  OR PhoneNumber

data OrderQuantity =
  UnitQuantity
  OR KilogramQuantity
```

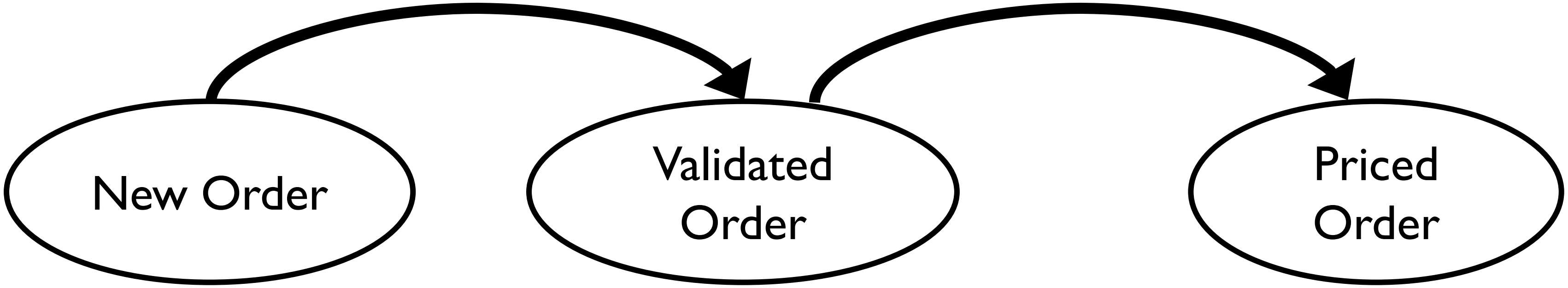
Document simple/constrained types

```
data UnitQuantity = integer between 1 and ?

data KilogramQuantity = decimal between ? and ?
```

Be aware of possible state transitions



Be aware of possible state transitions

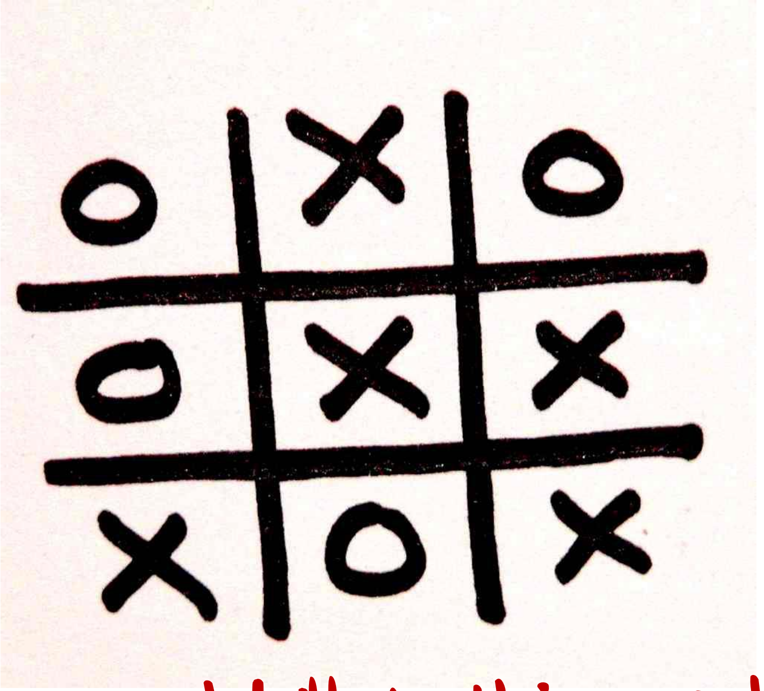


Exercises: You are the domain expert!

For each of the following domains, document the events and the associated data.

Exercise:

Tic-Tac-Toe / Noughts and Crosses



O	X	O
O	X	X
X	O	X

We'll do this one together

Group Exercise: Domain discovery

Pick a domain, then document the events and the associated data.

Cash Machine

Microwave

Coffee Machine

Delivery

Your own domain

Reviewing the models

- Discovery is hard
  - – Becoming a domain expert is harder!
- Goal: Building a shared model
  - – Did it work? Are you using the same vocabulary?
- Goal: Fast feedback on the design
  - – A few weeks coding can save hours of talking!
- Optional - Paste your domain models into the Google Docs file for group review

End