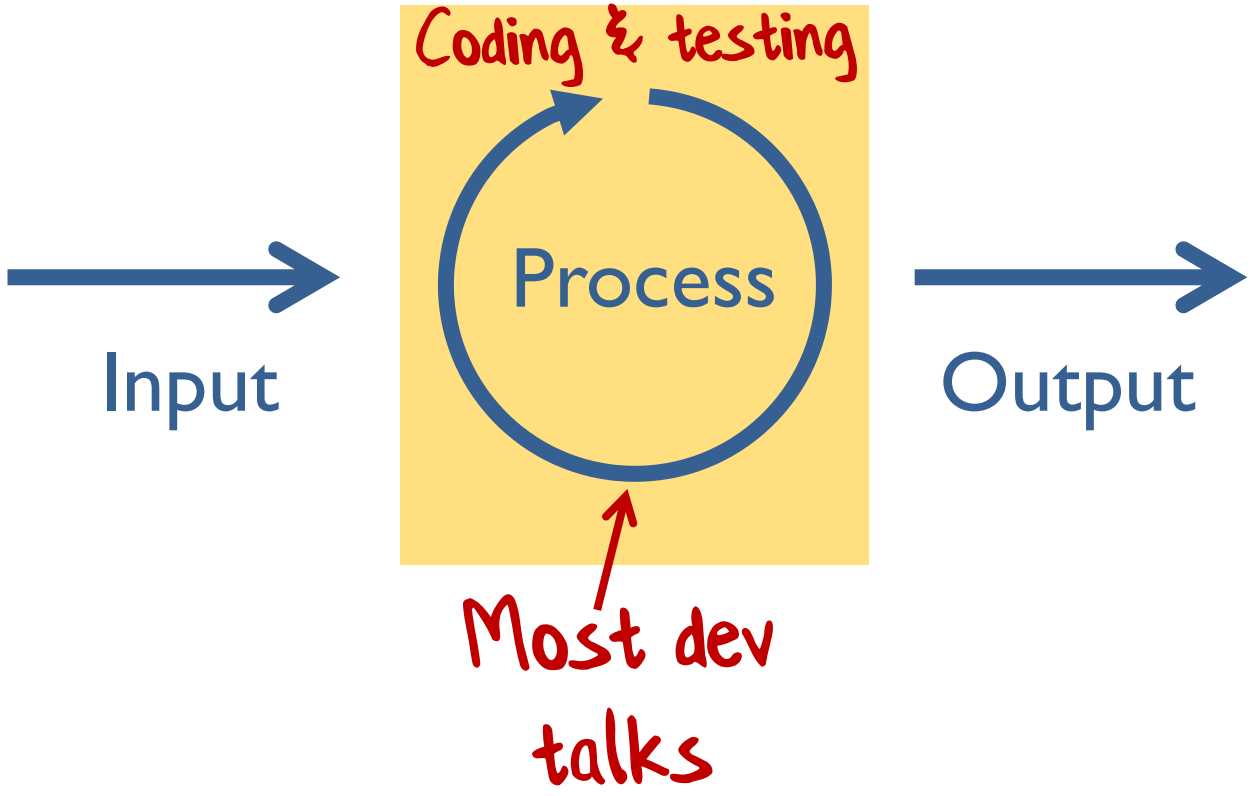
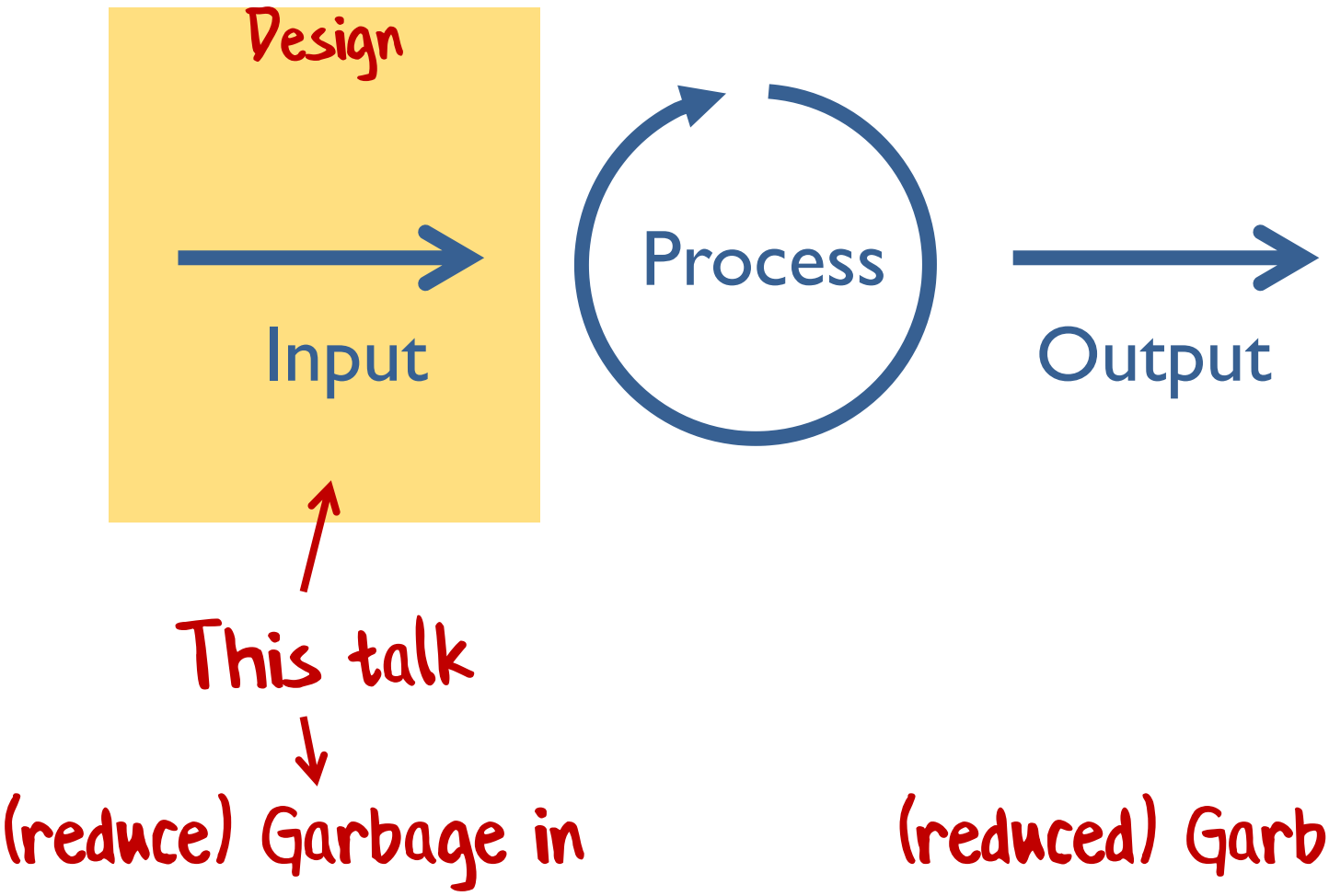
Domain-Driven Design and Domain Modelling

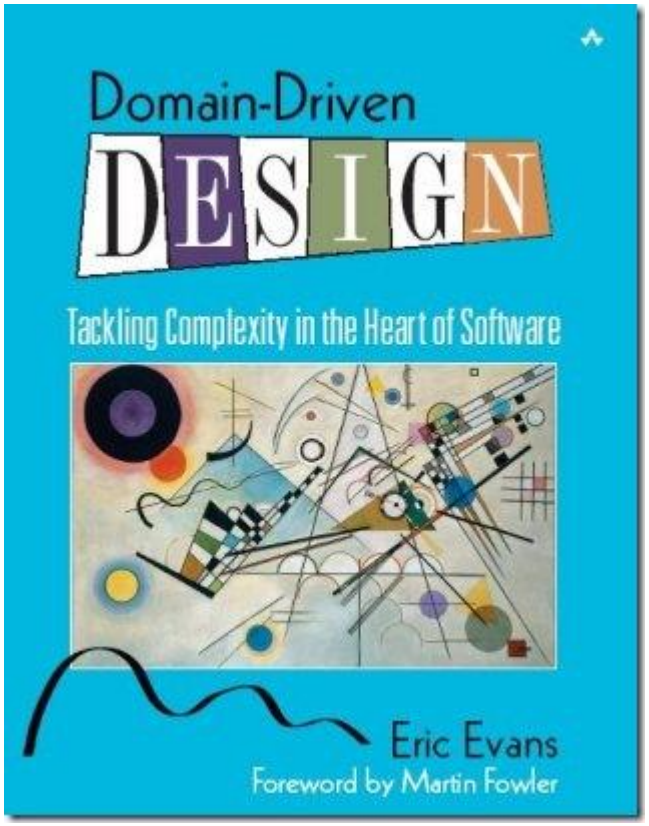
The software development process

The software development process



The software development process

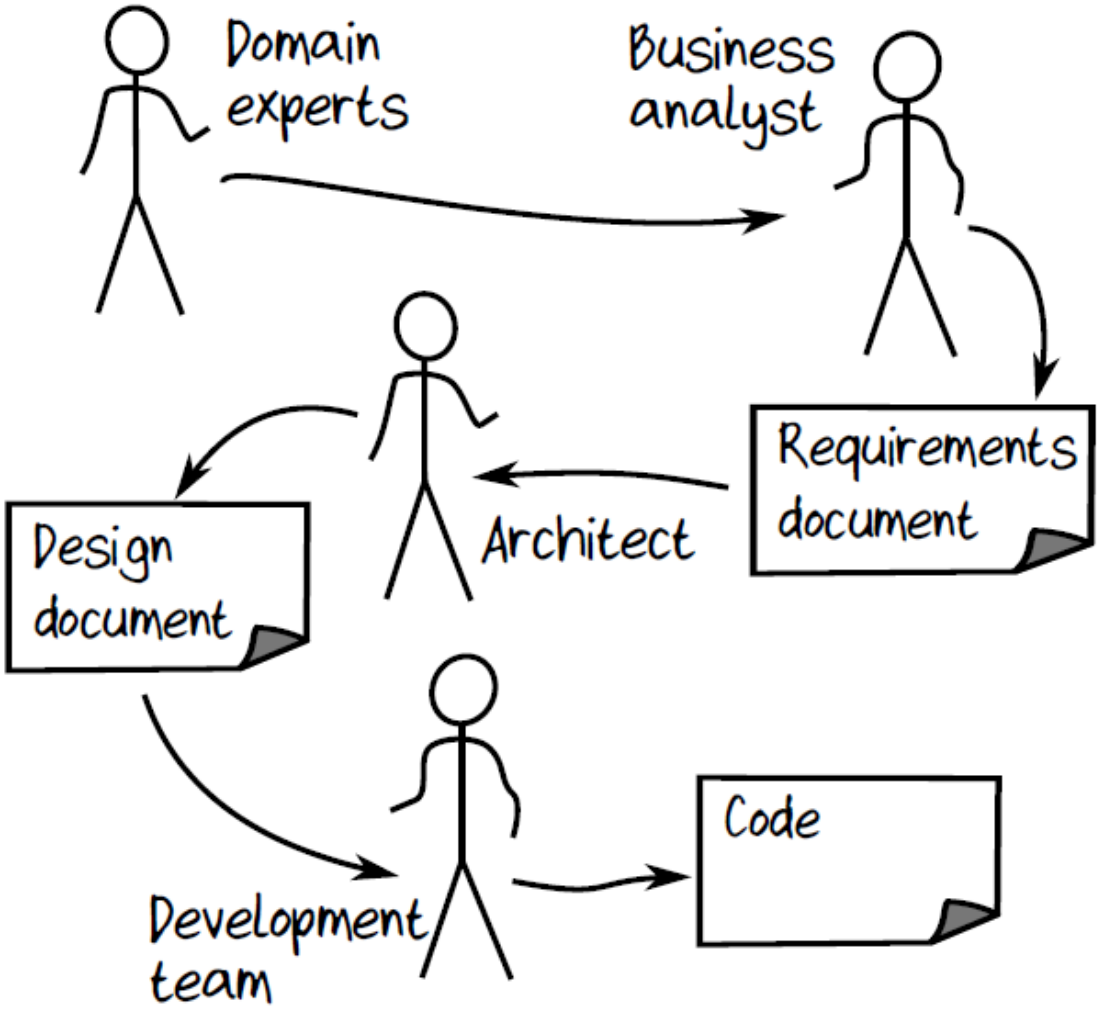




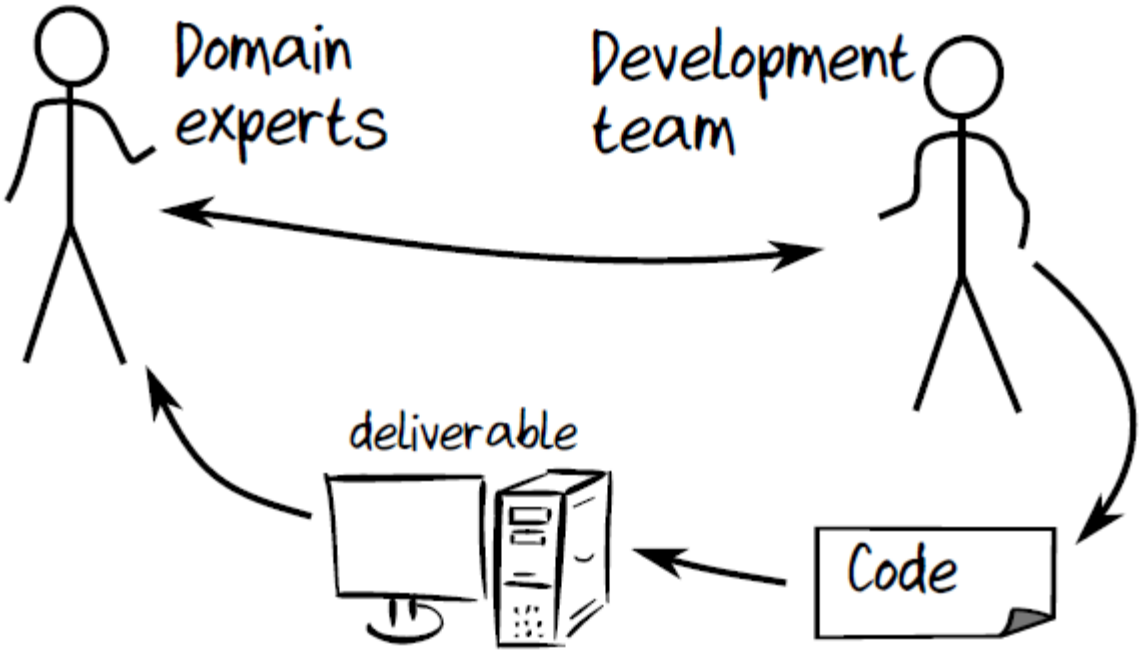
"Focus on the domain and domain logic rather than technology"
-- Eric Evans

Why Domain-Driven Design?

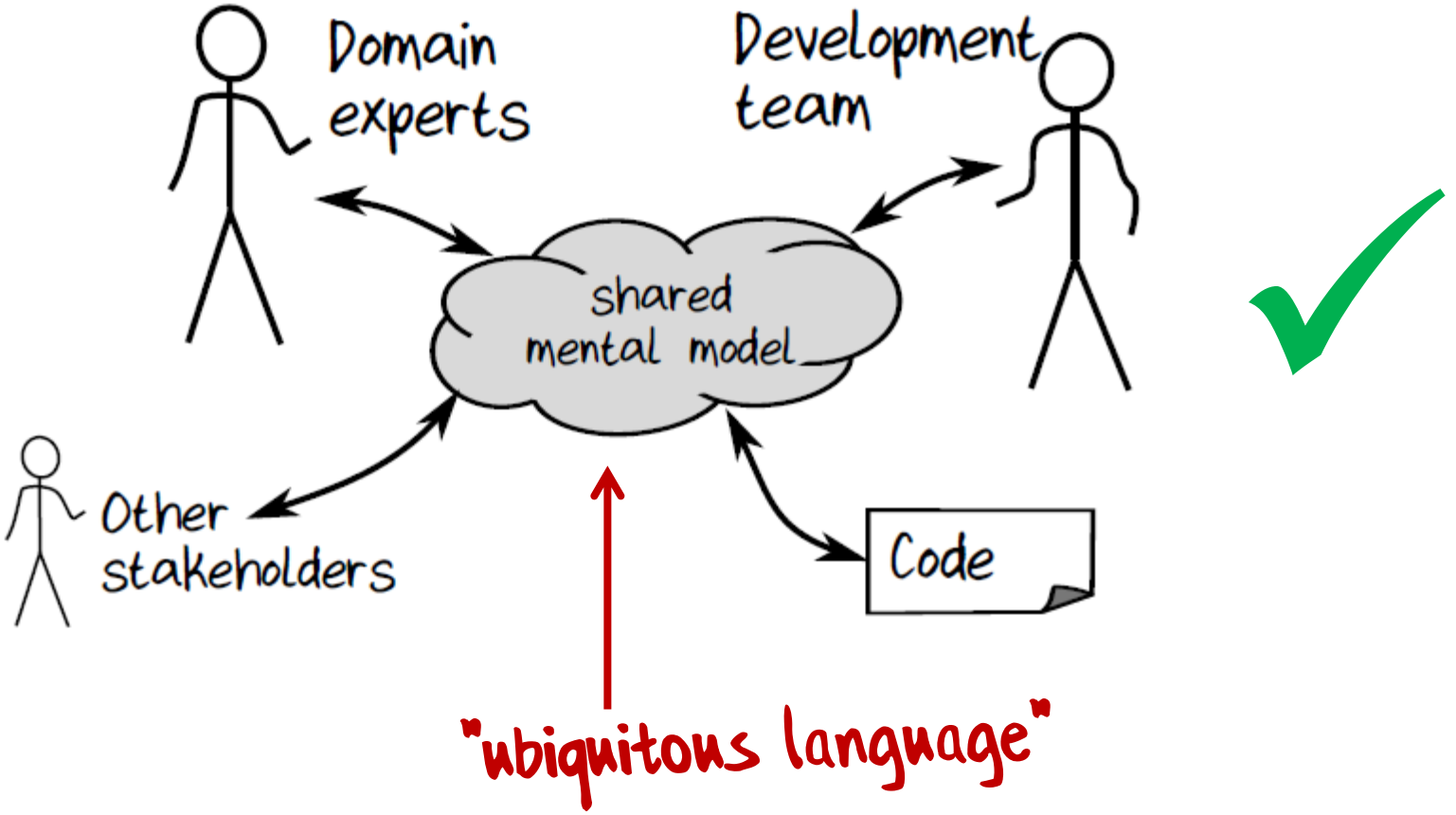
Waterfall


Domain experts
Business analyst
Requirements document
Architect
Design document
Development team
Code

Agile



DDD

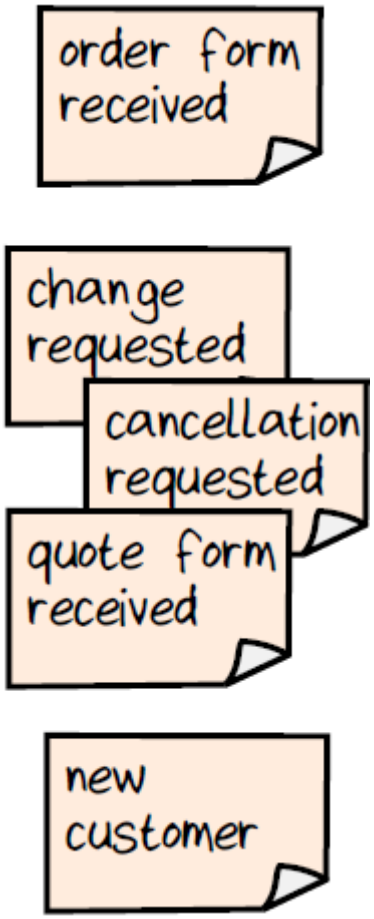

Domain experts
Development team
shared mental model
Other stakeholders
Code
"ubiquitous language"

Understanding the domain through business events

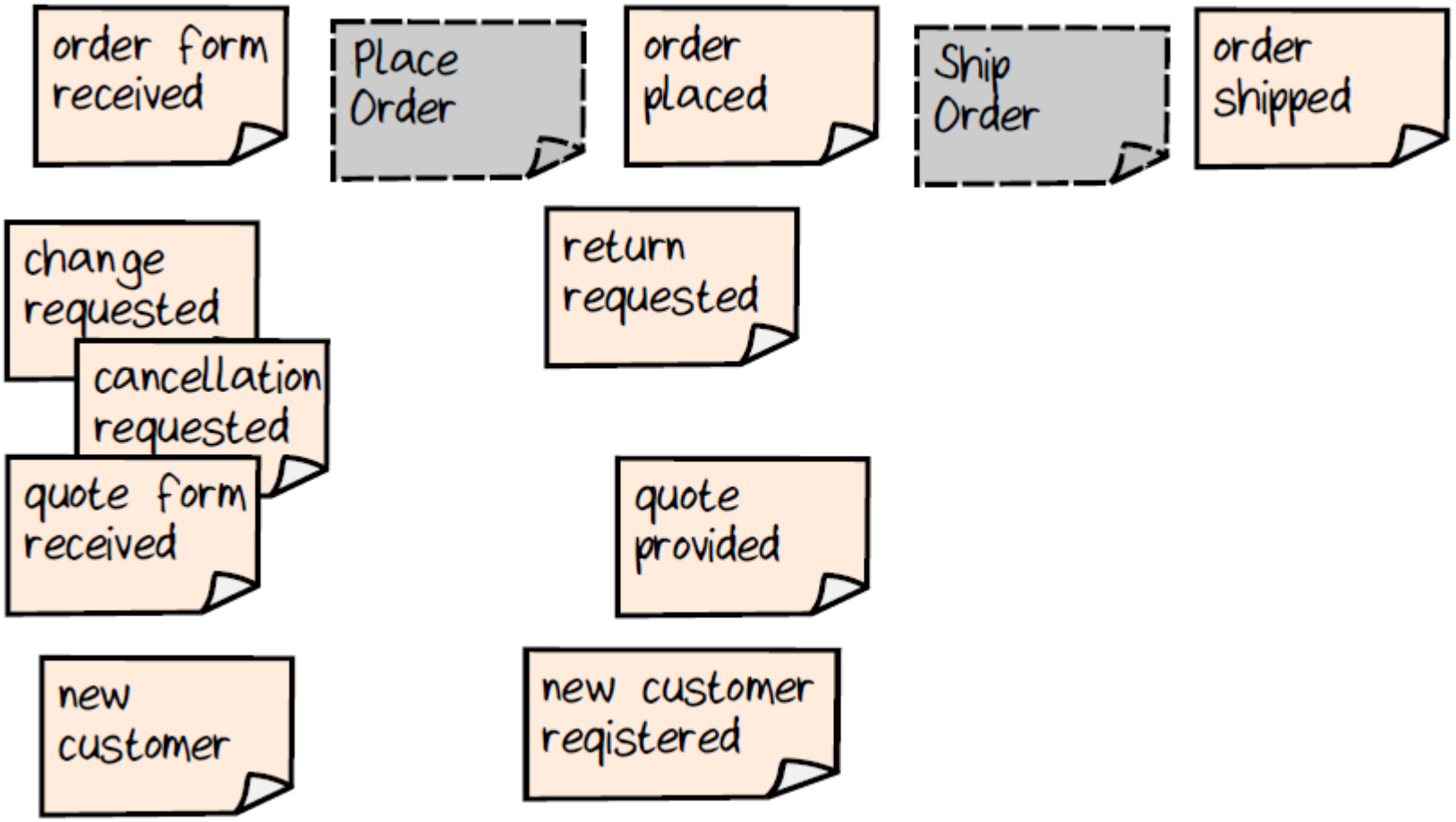
Why events are important

- 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
- What causes an employee/process to start working with that data and adding value?
  - An event!

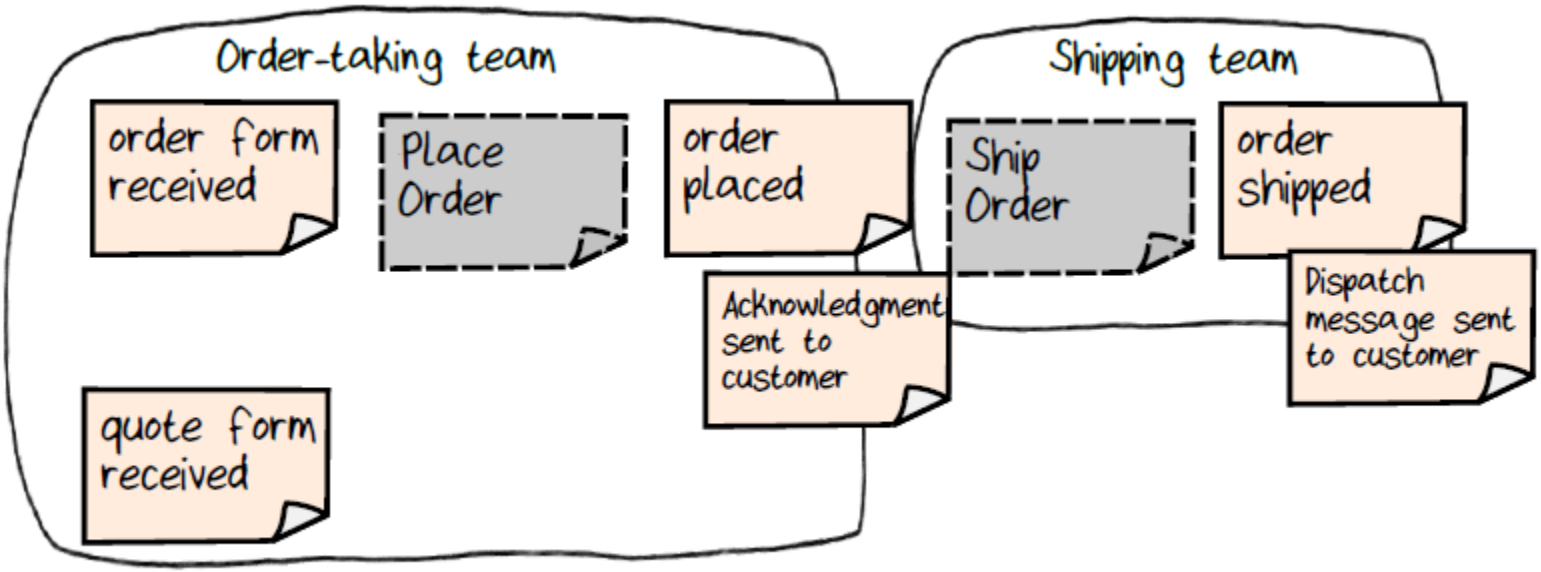
Finding the events with Event Storming



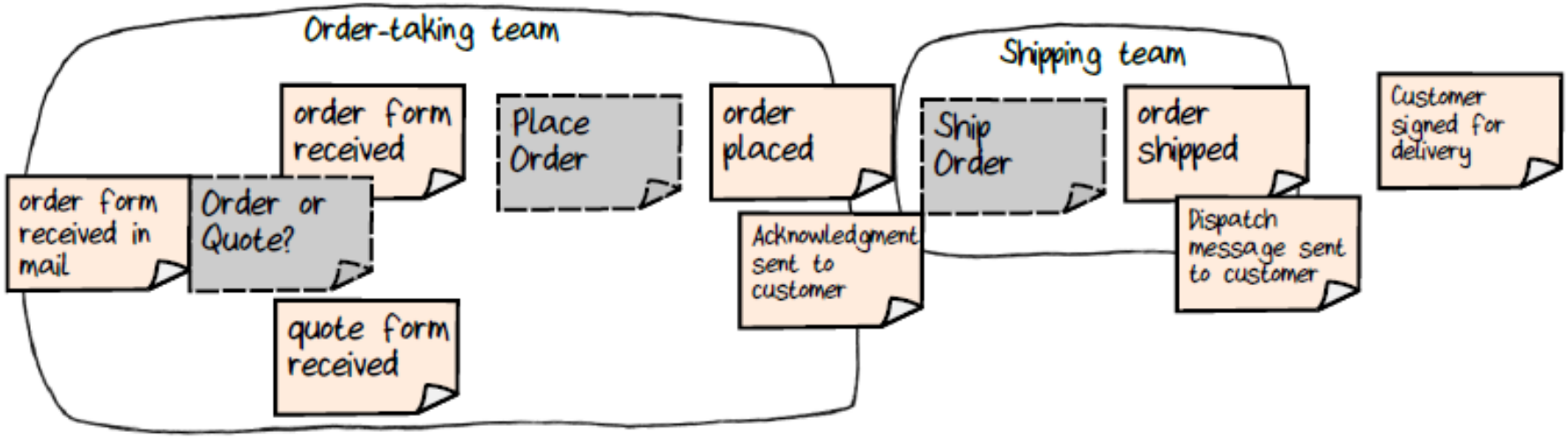
Finding the events with Event Storming



Then group the events

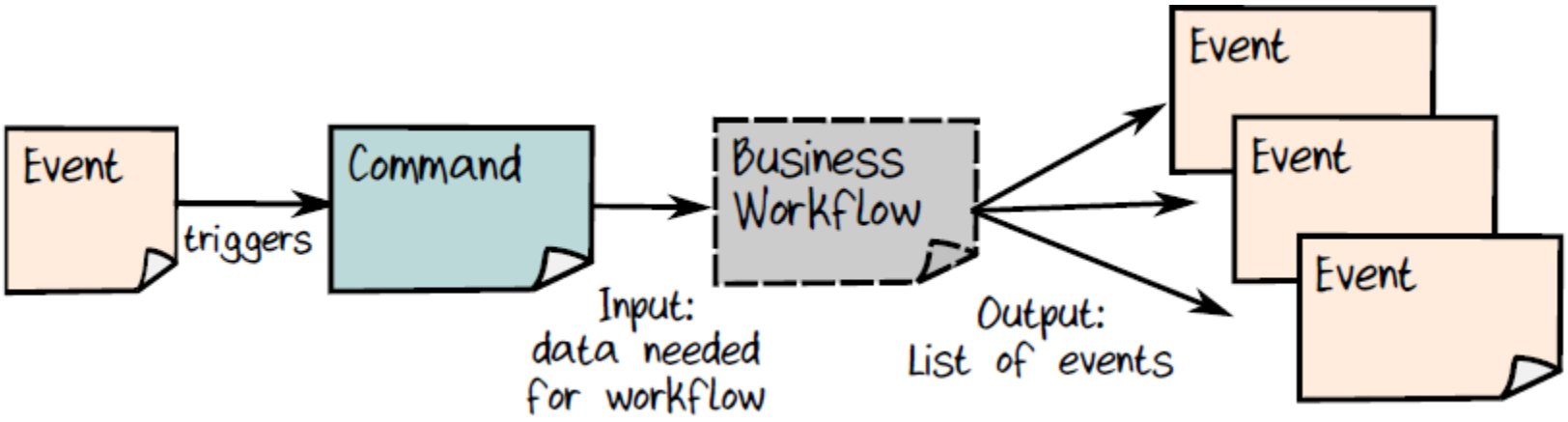


And extend to the edges



Introducing "workflows"

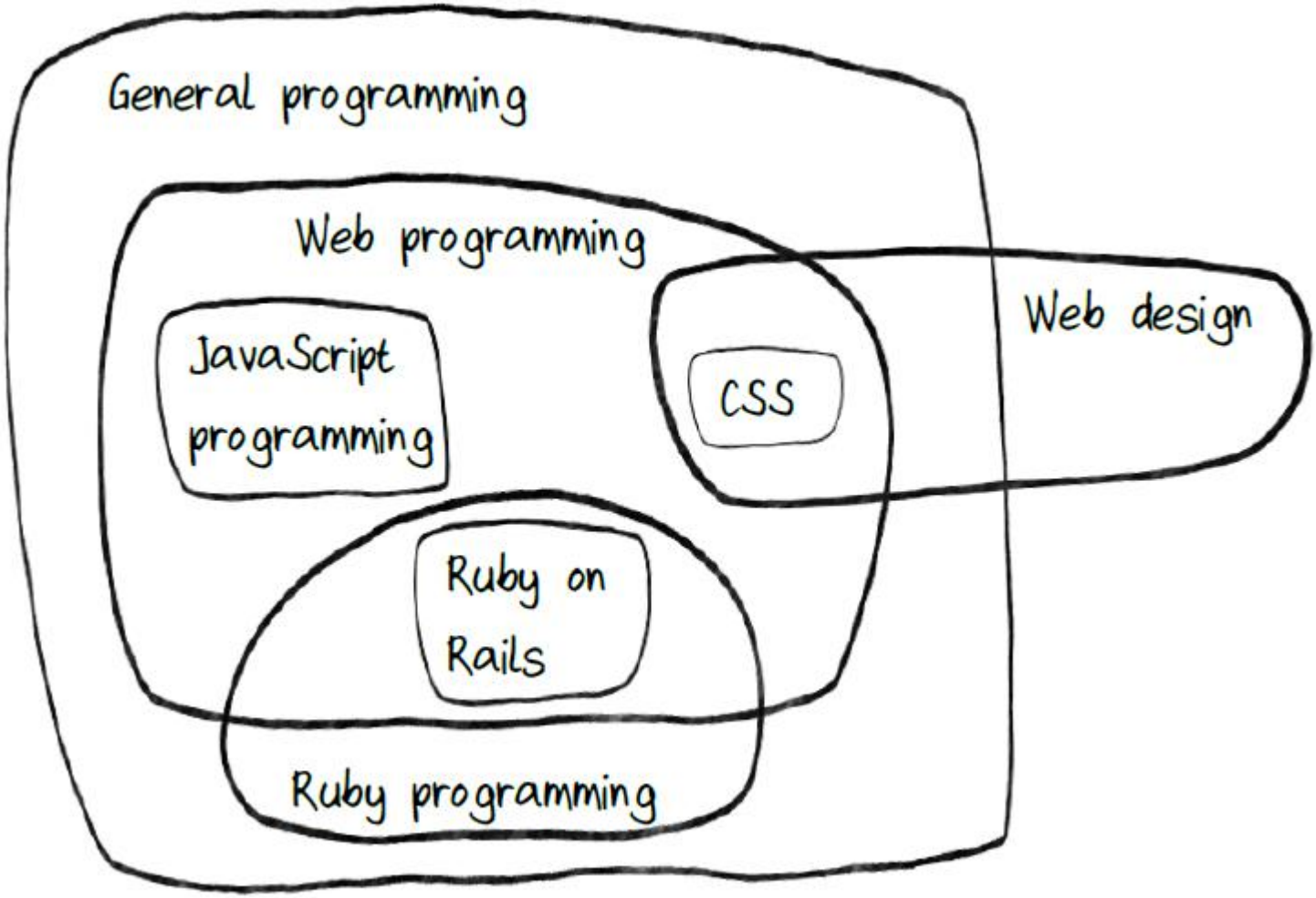
Events, commands, workflows



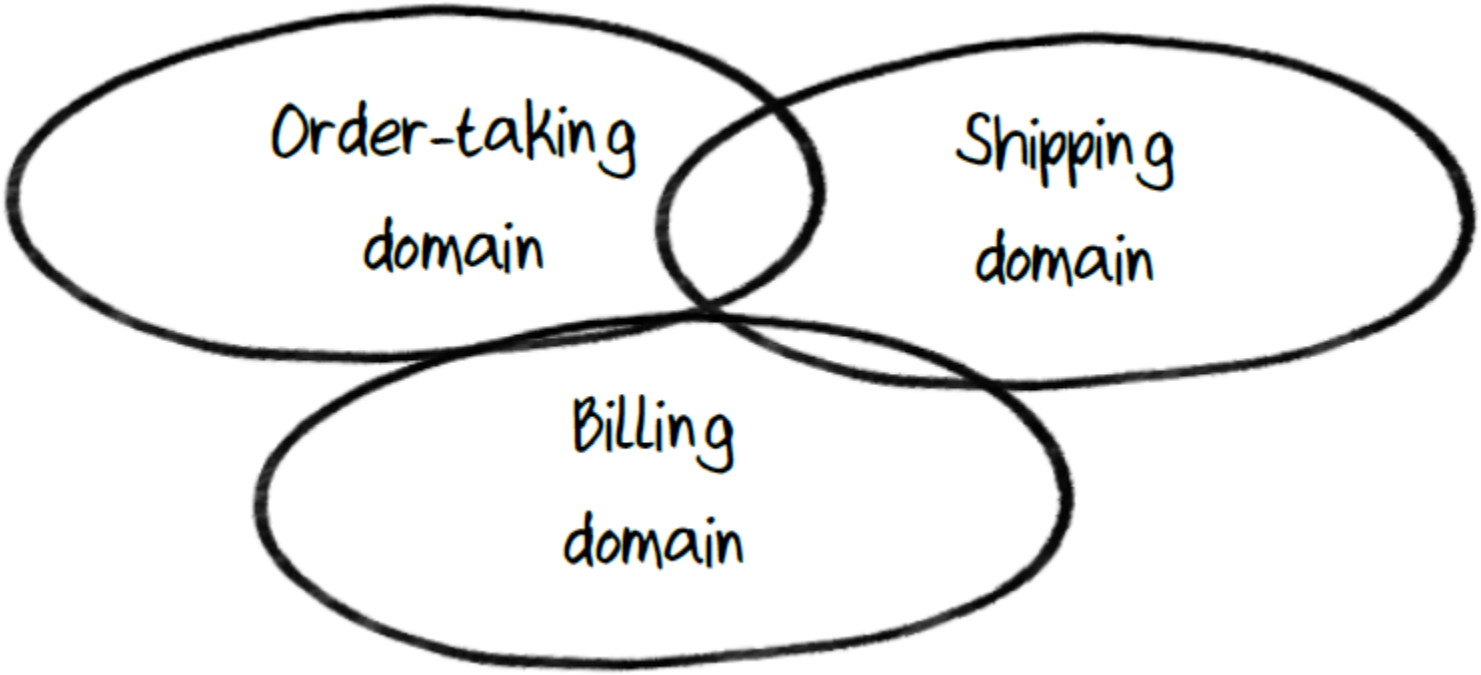
These are the units of work when coding

Introducing "subdomains"

What is a subdomain?



What is a subdomain?

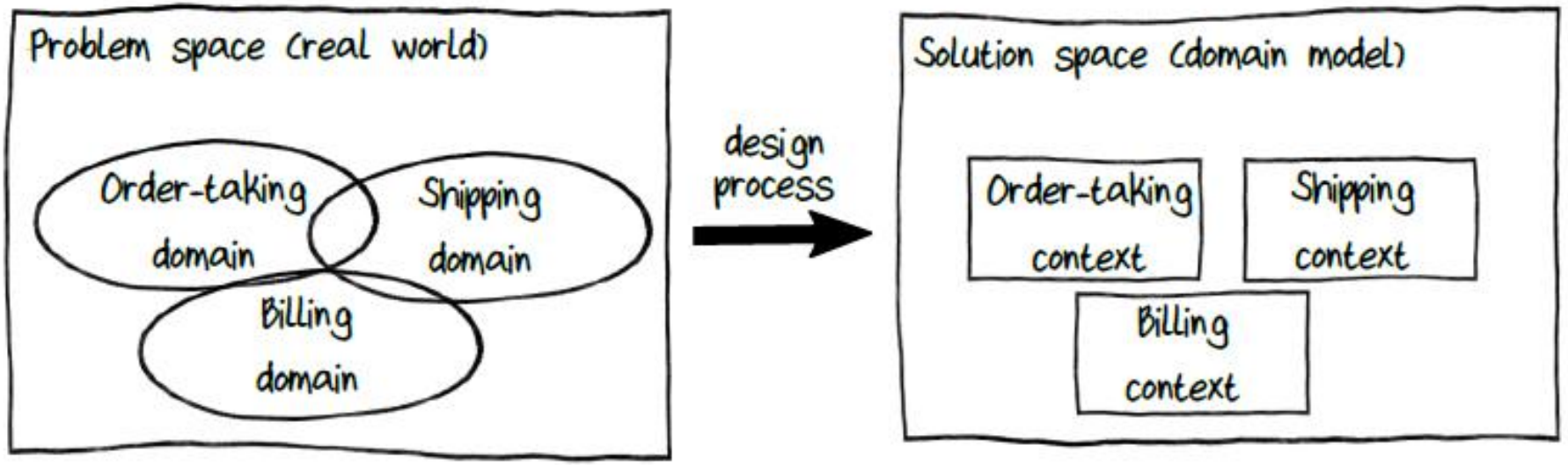


Introducing "bounded contexts"

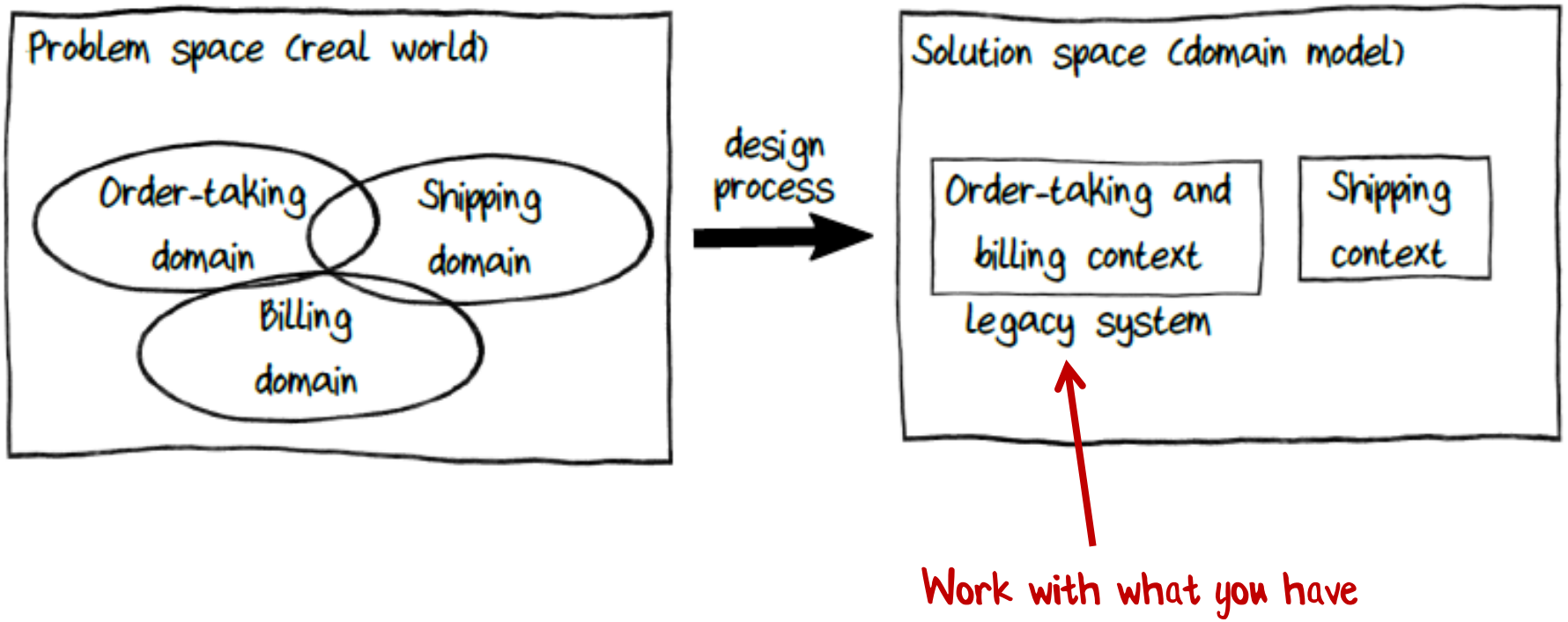
"Problem space" vs. "Solution space"

- 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!

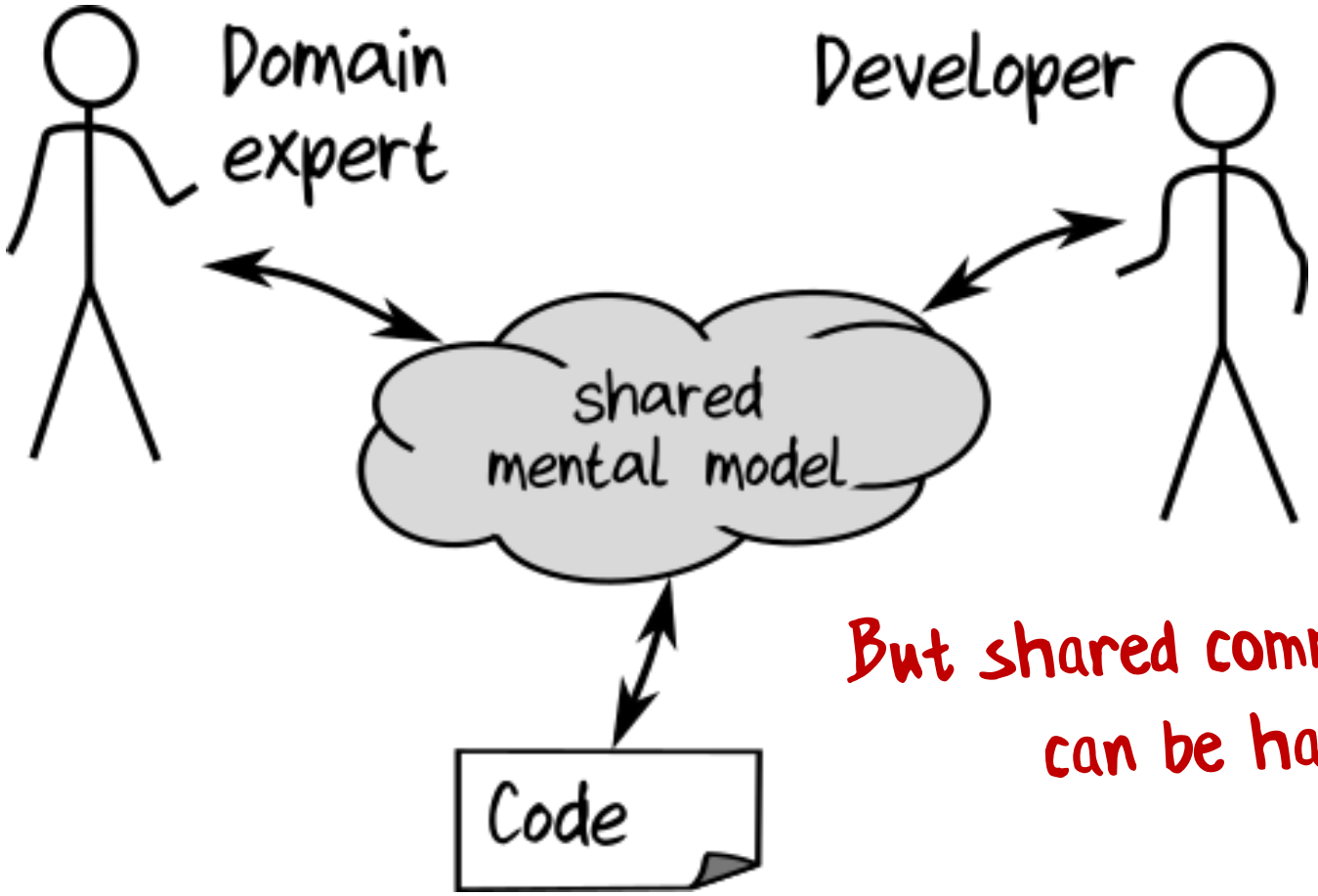
How to get the contexts right

- 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
  - 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

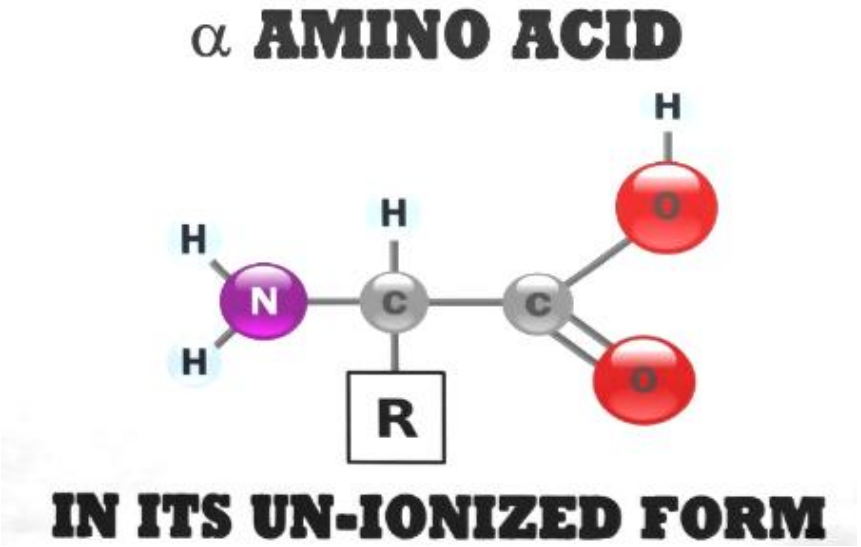
- The Ubiquitous Language is a set of concepts and vocabulary associated with the domain and is shared by
  - Domain experts
  - Development team
  - The source code
- The "everywhere language"

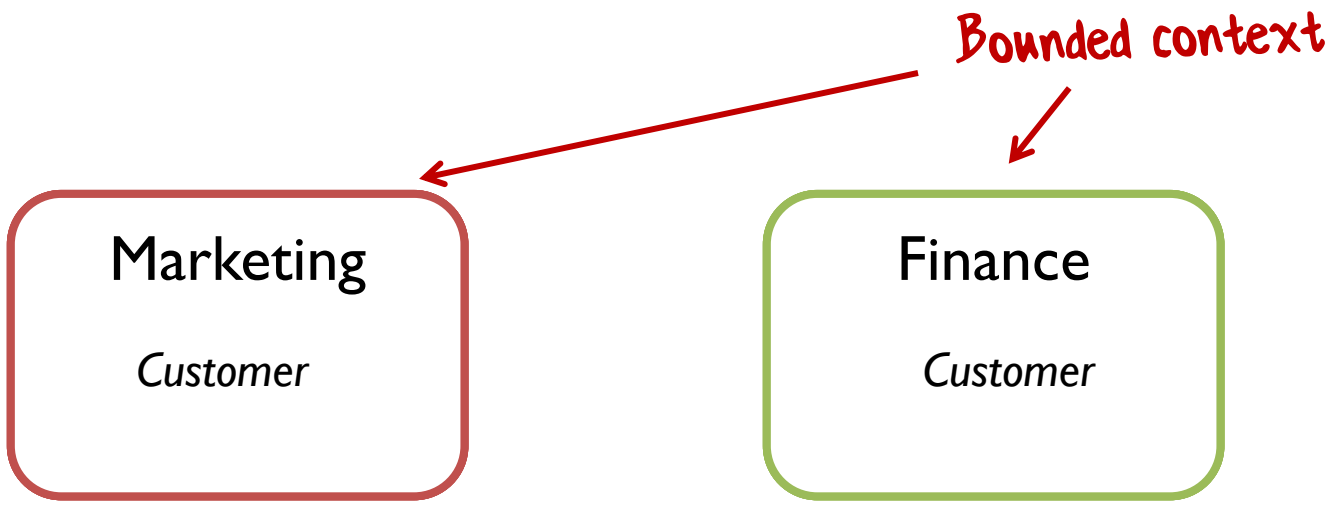


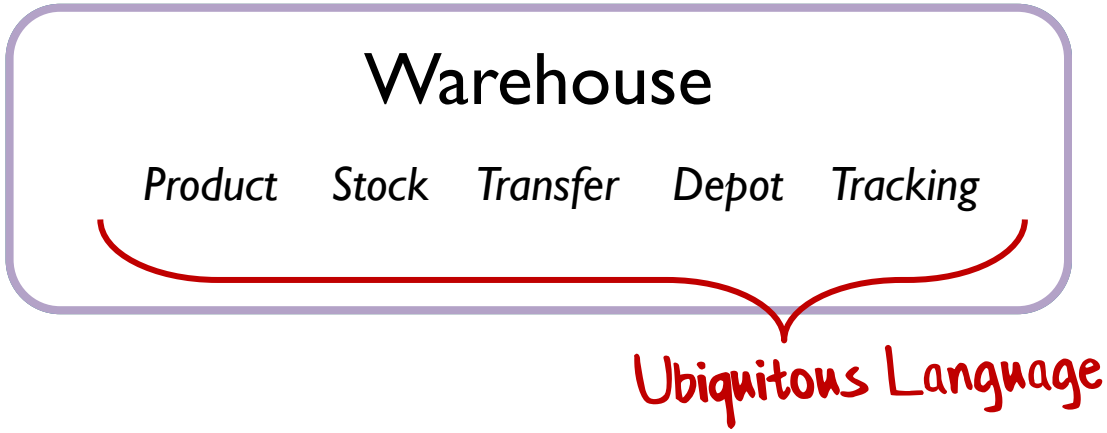
But shared communication can be hard!

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


BETTER
MEDICAL
SCHOOLS
GIFT PACKAGES


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


Bounded context
Marketing
Customer
Finance
Customer

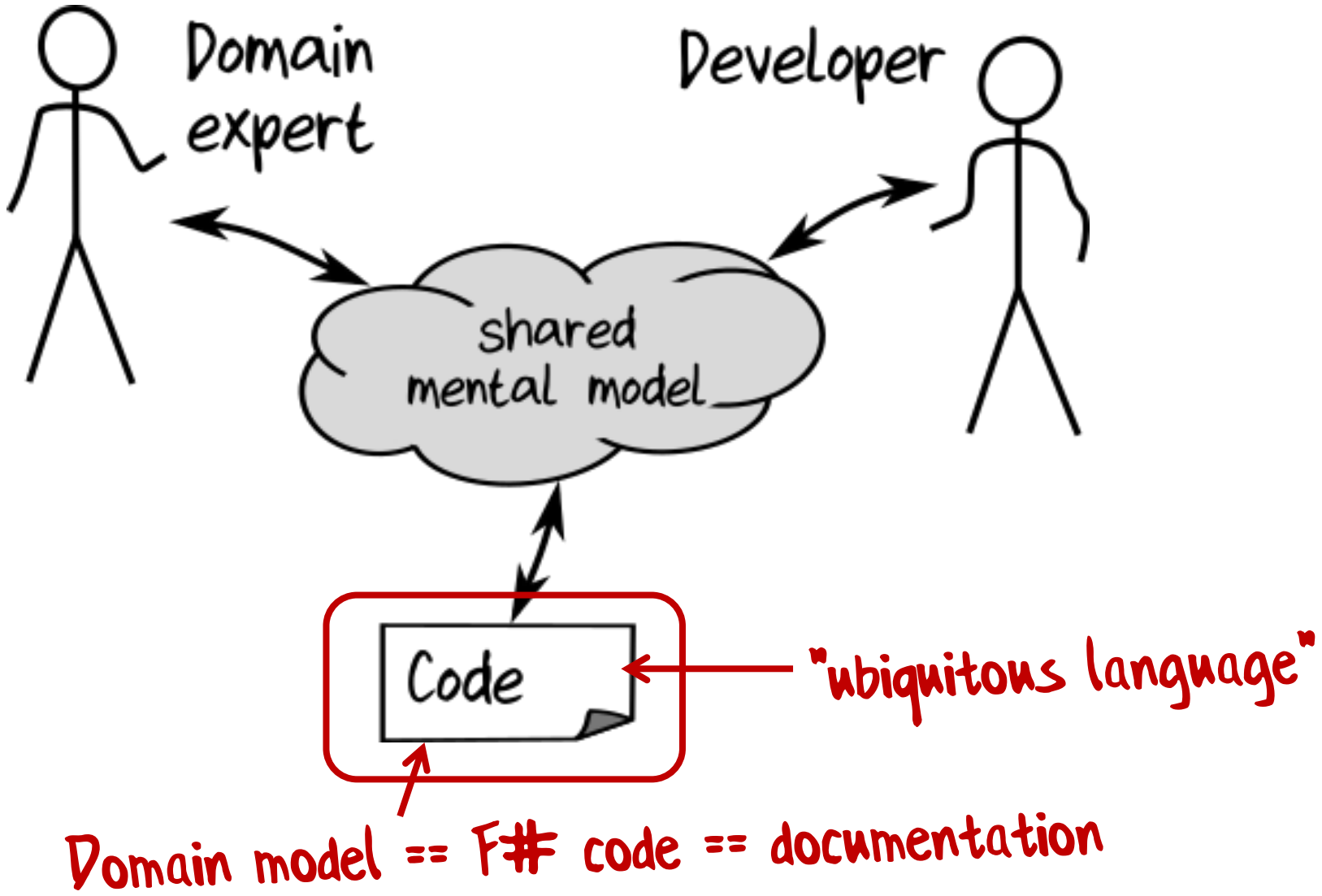

Warehouse
Product
Stock
Transfer
Depot
Tracking
Ubiquitous Language

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 | 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
```


Domain expert
Developer
shared mental model
Code
"ubiquitous language"
Domain model == F# code == documentation

```
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
```

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

'*' means a pair. Choose one from each type

list type is built in

X -> Y means a function
- input of type X
- output of type Y

```
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
```

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

Do you think a non-programmer could understand this?

```
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
```

"persistence ignorance"

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

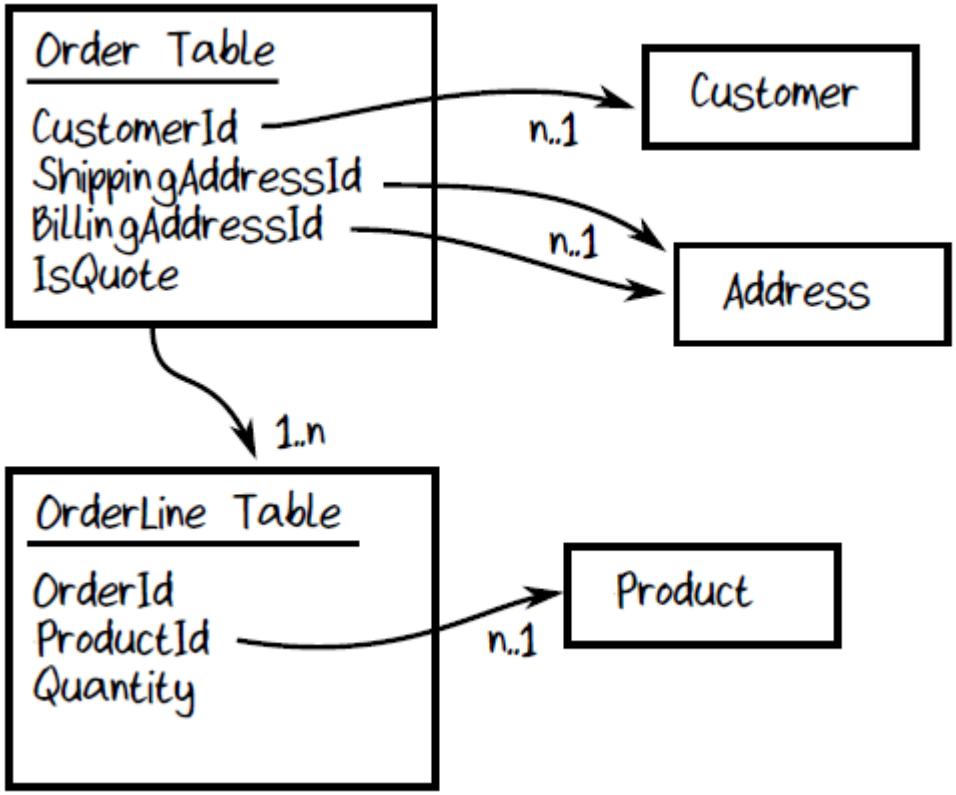
This is not pseudocode – this is executable code!

Summary of DDD concepts

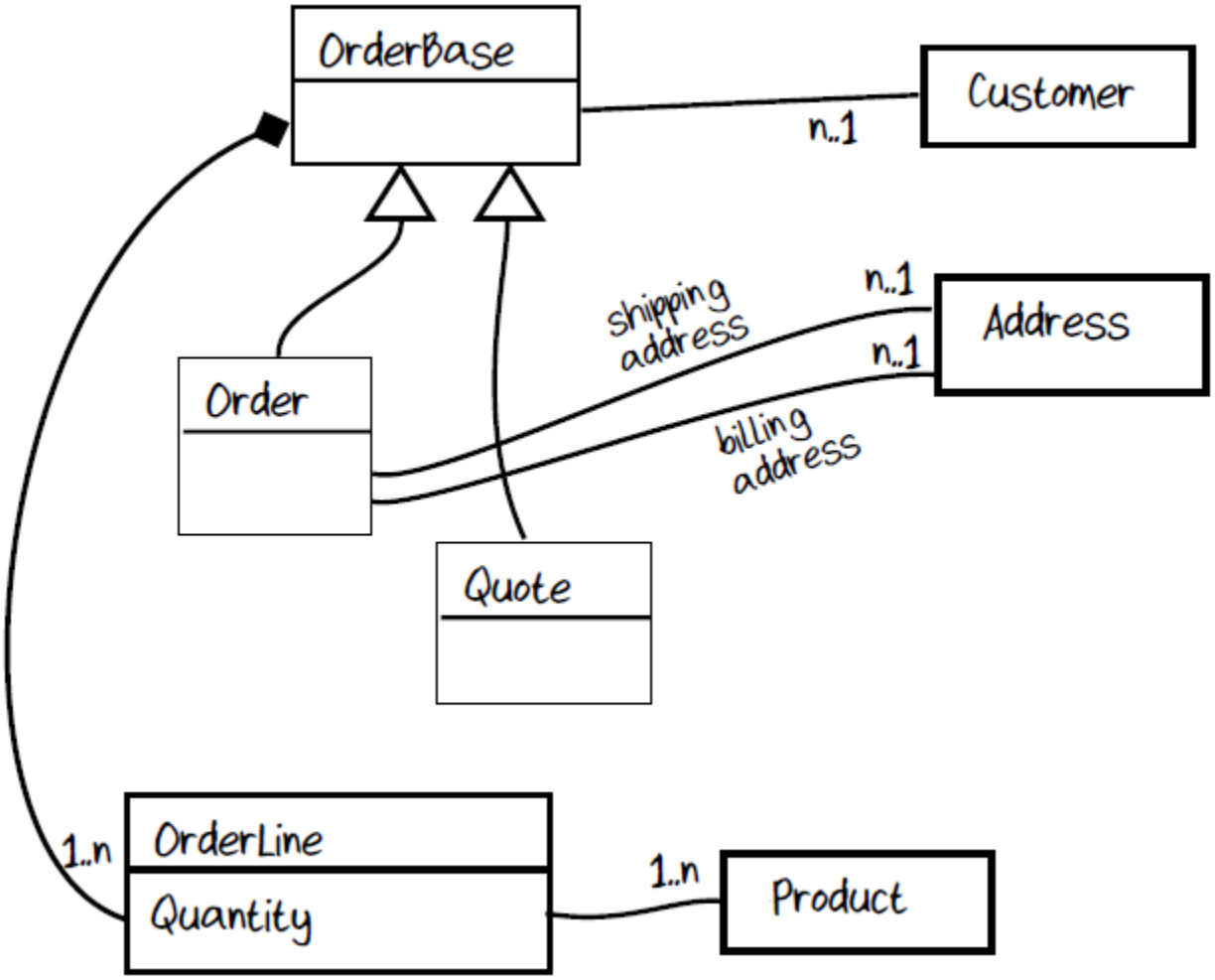
- Domain Model
  - A set of simplifications that represent those aspects of a domain that are relevant to a particular problem.
  - The domain model is part of the solution space
- Bounded context
  - A subsystem in the solution space with clear boundaries that distinguish it from other subsystems.
- Ubiquitous Language
  - A set of concepts and vocabulary associated with the domain and is shared by both the team members and the source code.
- Domain Event
  - A record of something that happened in the system. It is always described in the past tense. 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.

Getting started with domain modeling

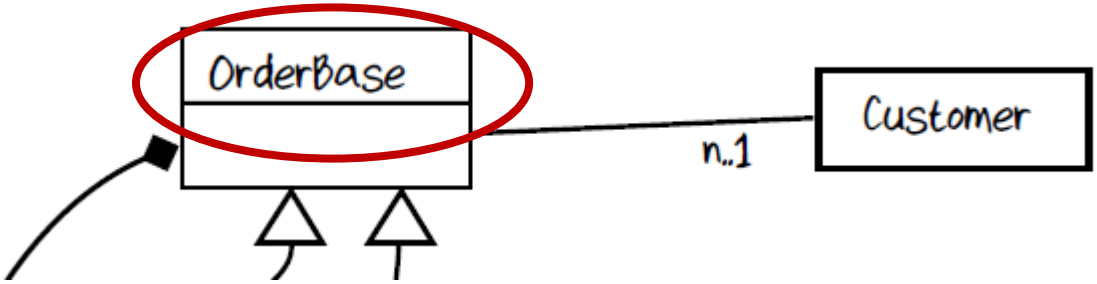
It's not database design!



It's not OO design!



It's not OO design!



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

A domain expert wouldn't know what you meant by these words.

My recommended way of domain modeling

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:
  Product catalog
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 Customer = FirstName AND LastName

data OrderLine = ProductId AND Quantity
```

Document choices with OR

```
data ContactInfo =
  EmailAddress
  OR PhoneNumber

data OrderQuantity =
  UnitQuantity
  OR KilogramQuantity

data UnitQuantity = integer between 1 and ?
data KilogramQuantity = decimal between ? and ?
```

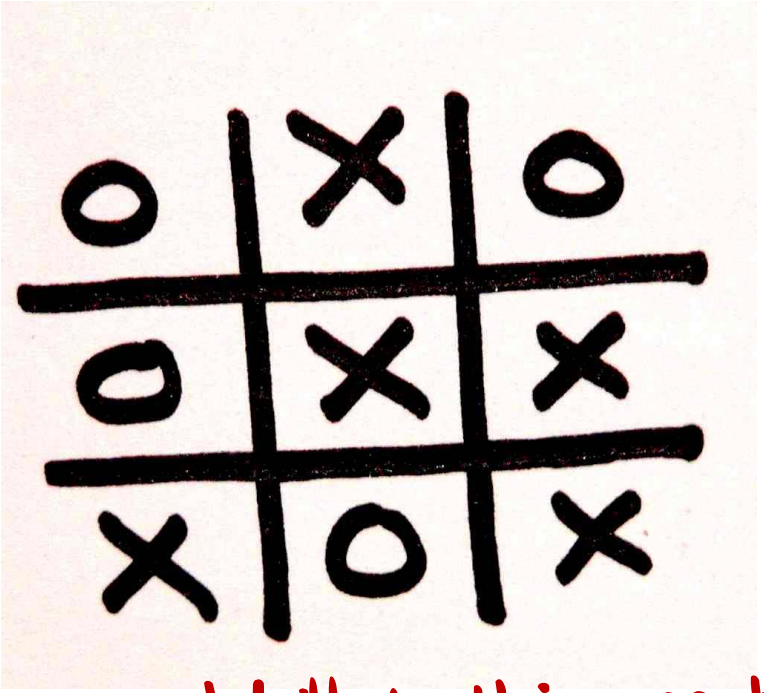
Exercises

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

Then paste your domain models into the GDocs file.

Exercise:

Tic-Tac-Toe / Noughts and Crosses / Kruisje-rondje



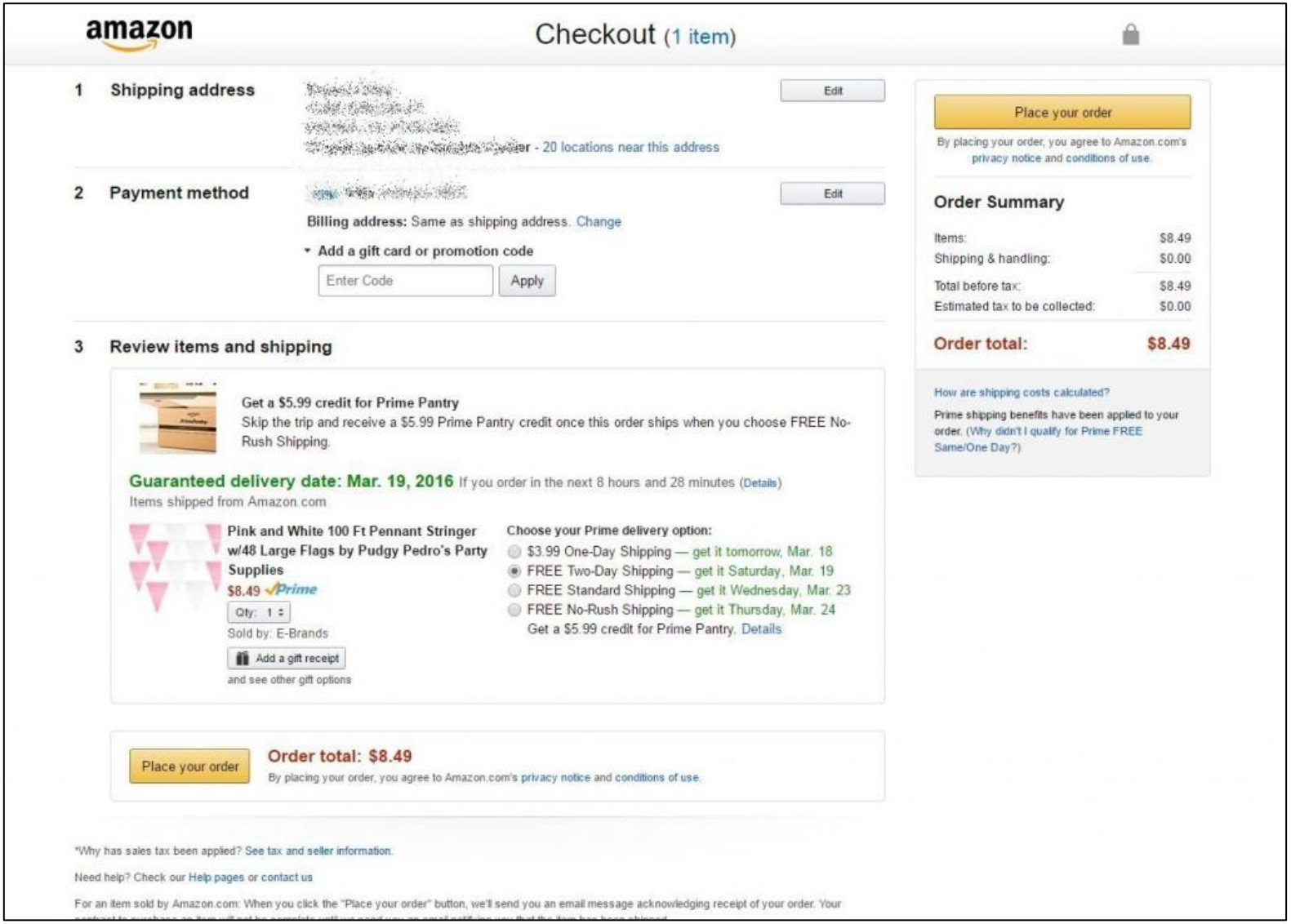
We'll do this one together

Exercise: ATM / Cash machine

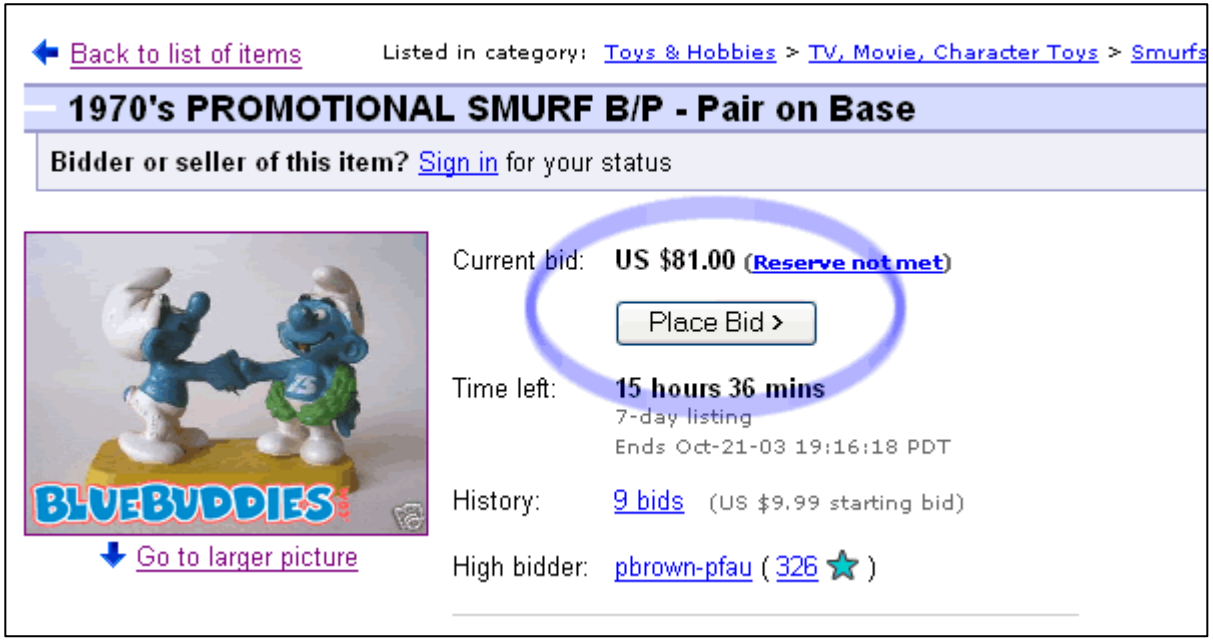
Exercise: Microwave

Exercise: Self-service coffee maker

Exercise: Amazon shopping



Exercise: Ebay-style bidding



Exercise: Your own favourite domain

End