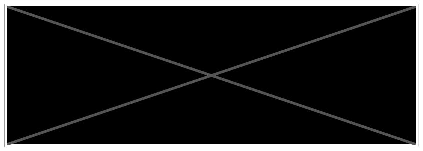
Programming 6

Data, CQRS, Event Sourcing

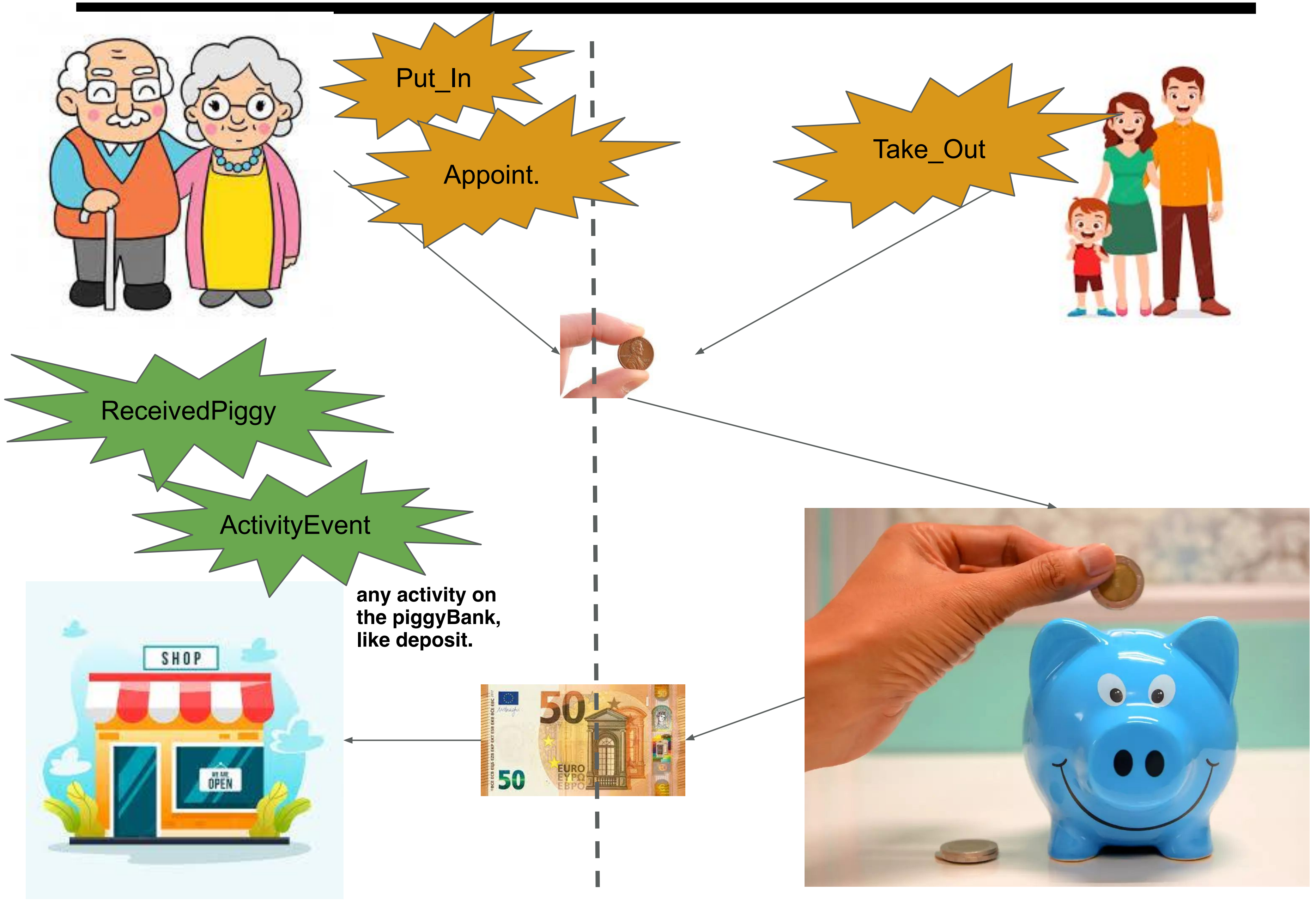
Text



Menu

- The Idea behind Event Sourcing
- The Idea behind CQRS
- Async vs Sync communication

Putting it all together



First: a disclaimer …yet again…

Event Sourcing

Idea: Knowing where you are, is sometimes not enough, sometimes you also want to know how you got there.

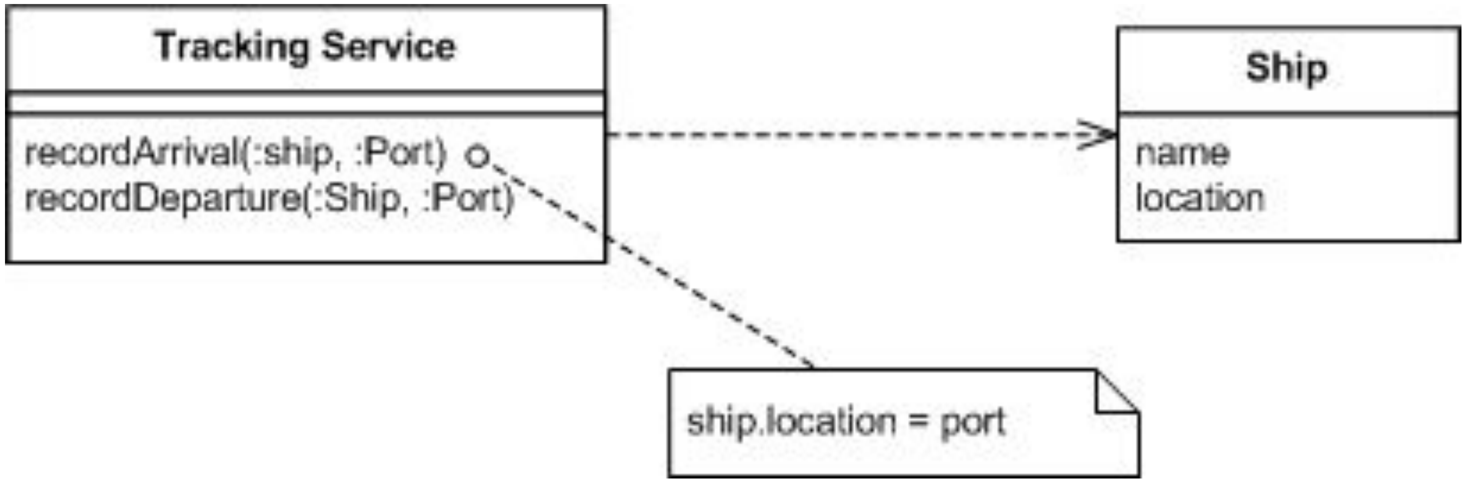
Event Sourcing stores all changes to an object state as a sequence of events. By replaying those events, you can reconstruct a paste state.

If you replay all events, you'll get the current state.

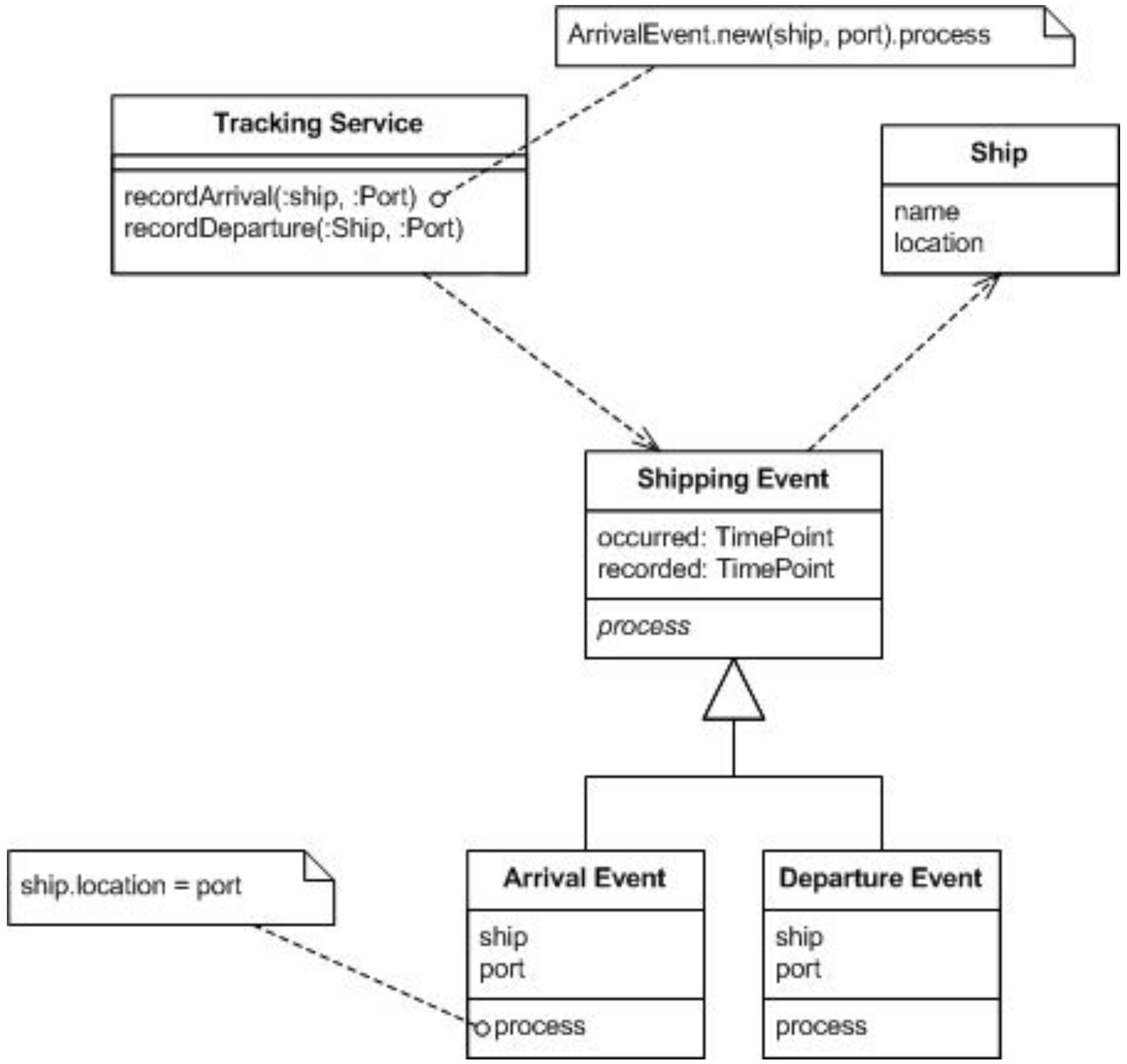
Key is that all changes to the domain object is tracked by an event.

Data, data, data -> for insights, auditing, ….

Event Sourcing



Event Sourcing



Event Sourcing

How to get the current state of a domain object?

- Get a “Blank” domain object
  Then replay all events on top of it

- Take a snapshot every x-number of events and instead of using a blank domain object, use the snapshotted domain object to replay all events.

- Store the current application state and the events log. Only replay when the event sourcing functionality comes in handy.

* Do not share databases! If an external system needs to read or write… let them pass through a gateway or adapter.

* Event versioning!

* What if our domain object state is not correct?

Goes very well with CQRS!

Good candidates: when to use?

In theory every domain aggregate change might be interesting for event sourcing.

If it can be done with a CRUD application, you should opt to implement a CRUD application.

Event sourcing becomes interesting when you get questions like:

“I’d like to have a monthly overview of our budget”

“Can you check what people are regretting the most adding to their shopping cart?”

“Can you add an approval workflow that can remove actions made to the batch”.

-> again, it all boils down to get to know your business domain!

What domain will be the most obvious to consider? YOUR CORE DOMAIN!

Our PiggyBank!

“What frequency get’s Bobby his allowance from his parents”

“What is the biggest donation Bobby received from his grandparents?”

“What is the best month to be Bobby!”

“When does Bobby buy his candy using money from his piggybank?”

Event Sourcing to the rescue.

Instead of keep the balance up to date.

We save the activities on the piggybank.

Our PiggyBank!

Enter 2 events:

PiggyBankPutMoneyInEvent.

PiggyBankTakeMoneyOutEvent.

Save these events.

Replaying these events gives you the current balance.

PiggyBankPutMoneyInEvent: 10€

PiggyBankPutMoneyInEvent: 10€

PiggyBankTakeMoneyOutEvent: 5€

PiggyBankPutMoneyInEvent: 10€

=> Current Balance= 25 €

Our PiggyBank!

PiggyBank has an ActivityWindow containing all activities on the PiggyBank.

You can use the Activity Window to get all activities within a time period. Can be saved as such to the database.

PiggyBankActivity: he said to use his/this approach to make things easier.

- Amount: The amount of the activity
- Action: The action of the activity (take out or put in)
- PointInTime: The time the activity took place.

he said to just use UIDS, never think you can do better job than java people :(

getBalance:

- Replays all activities within that ActivityWindow

Our PiggyBank!

ActivityWindow can then be used later on as a tool to limit your view on all the activities.

Query the PiggyBank with an ActivityWindow from 01/01/2022 to 31/03/2022.

```
public Optional<PiggyBank> loadPiggyBank(UUID piggyBankUUID, LocalDateTime start,
LocalDateTime end) {
```

After a couple of years your activities can grow large, you can sum them on the database or snapshot them.

Let’s introduce some concepts of CQRS.

About snapshotting

I did not understand this properly

Enter a BaseBalance that has snapshotted all the x-events.

this is a technical consideration, you will take the infra that can handle this.
in biggyBank, we can take snapshot before shopping.

When to snapshot?

choose wisely depending on your use case, some strategies:

- each period (period aligned with business)

  -> steer this period by using the passage of time event

- special event occurred?
- each N events (with 1 as smallest possible value for N)

-> added the snapshot in v4 of the piggybank.

Used the special event as strategy here.

CQRS

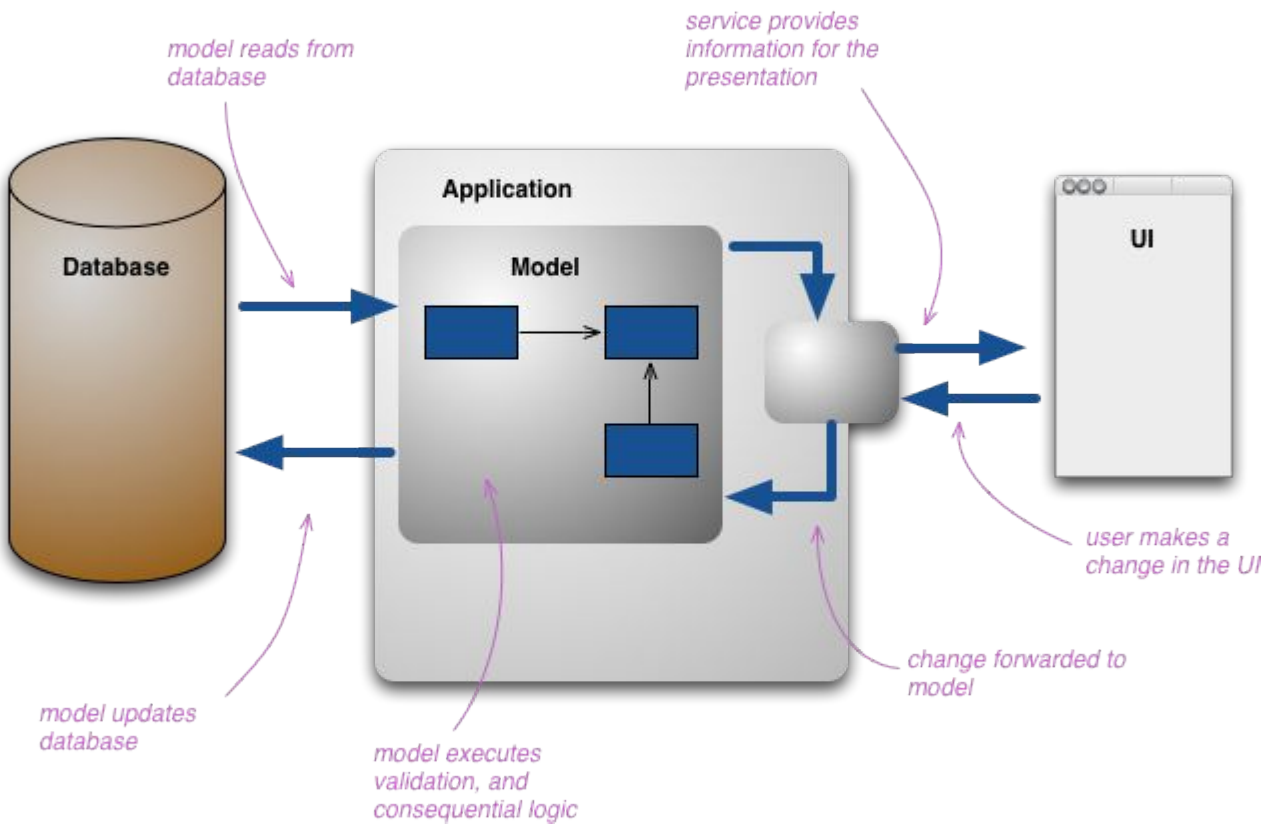
CQRS - Command Query Responsibility Segregation

Idea: The way you write data, is not necessarily the same way you'd like to read this data.

CRUD

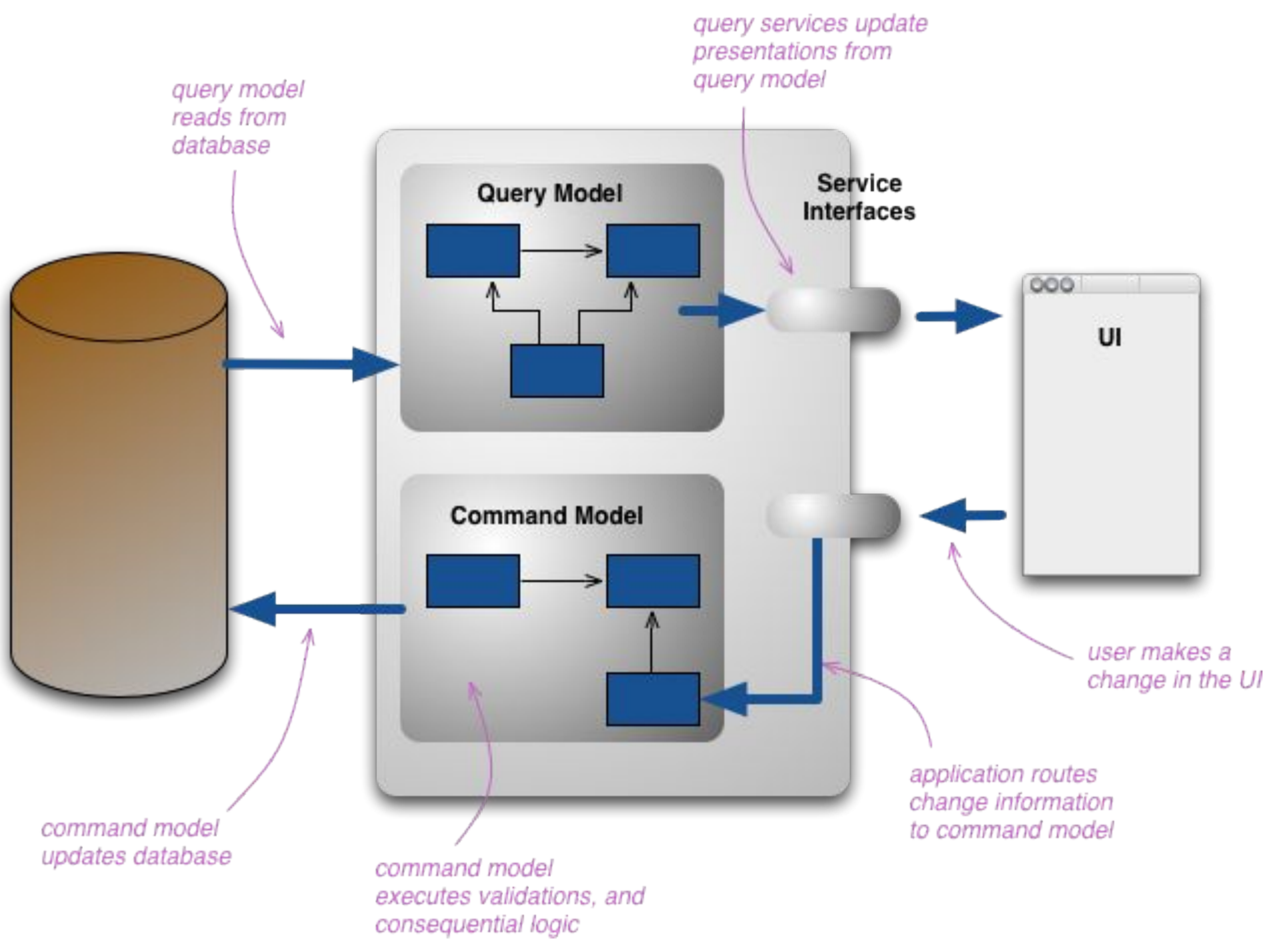
CRUD

We use the same record structure to store (create) those records, read, update or delete them when we are done with them.


model reads from database
service provides information for the presentation
Application
Model
Database
UI
user makes a change in the UI
change forwarded to model
model updates database
model executes validation, and consequential logic

CQRS

We create various representations of the information. Sometimes we want to combine information in one record or create virtual records combining information from different sources.


query services update presentations from query model
query model reads from database
Query Model
Service Interfaces
UI
Command Model
user makes a change in the UI
application routes change information to command model
command model updates database
command model executes validations, and consequential logic

Multiple representations

You already have multiple representations of your data

- Domain model (with different rules)
- Database model (with different rules)

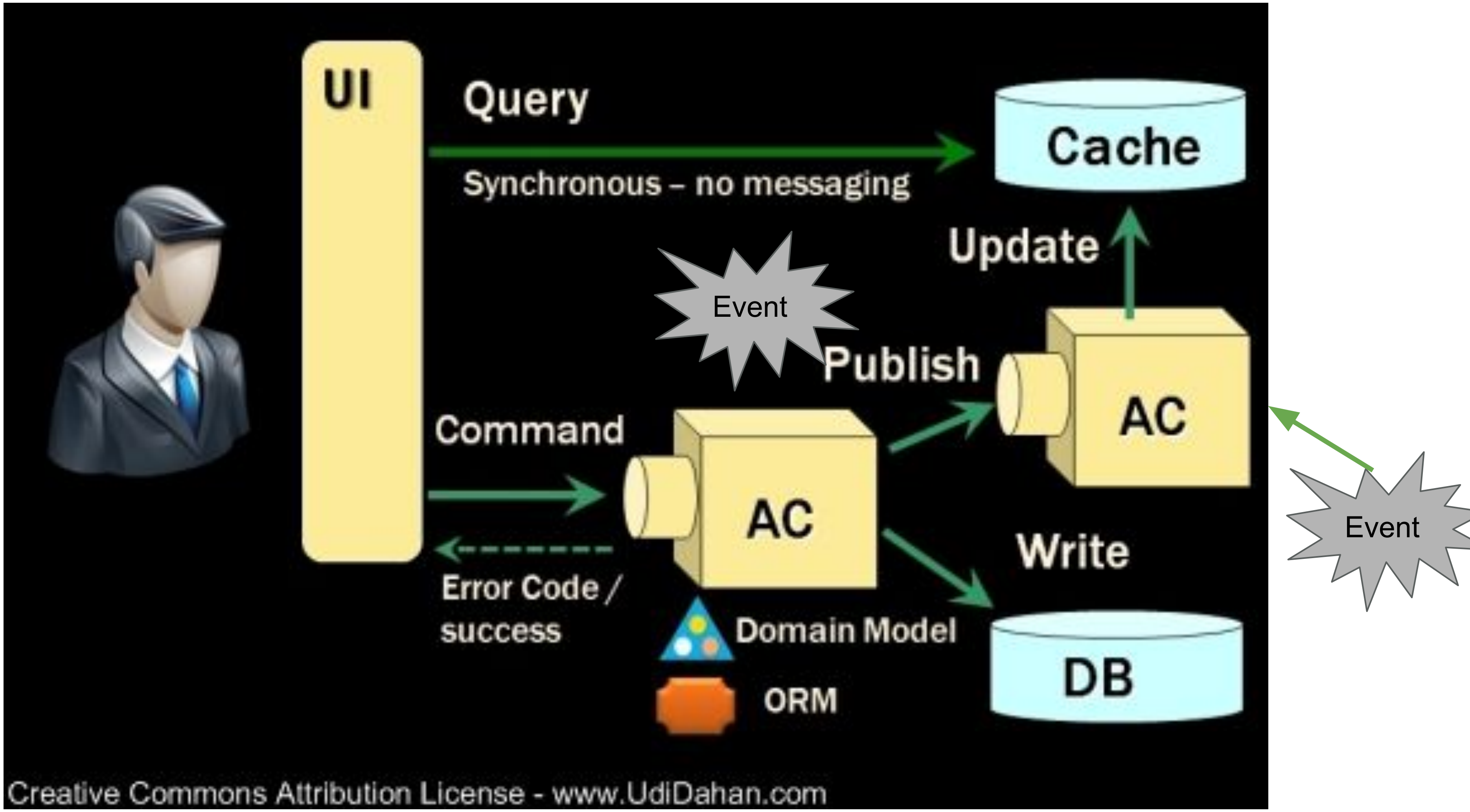
It has advantages and disadvantages to use this approach.

- (-) **It will** make your solution more complex
- (-) CQRS done on smaller parts -> no whole system (BC’s)
- (+) The presentation will fit the user’s needs
- (+) Writing can be very performant, asynchronous
- (+) You can use different infrastructure for reading and writing
- (-) But eventual consistency when doing so

-> CQRS fits very well with event driven architecture and complex domains… the same domains that are particularly well suited for a DDD approach

Conceptual

AC - Autonomous Component



Our PiggyBank!

Grandparents, interested in events:

PiggyBankPutMoneyInEvent.

PiggyBankTakeMoneyOutEvent.

Not interested in all events per sé, but interested in when currentbalance > 50

Save them and project them in the database with current balance, then check balance.

```
public Optional<PiggyBank> project(PiggyBankActivityCreatedEvent  event) {
   Optional<PiggyBank> piggyBankOptional  = piggyBankProjectionPort .loadPiggyBank(...)

   switch (PiggyBankAction.valueOf(event.action())) {
       case PUT_IN -> piggyBank.setCurrentBalance(piggyBank.getCurrentBalance() + 
event.amount());
       case TAKE_OUT -> piggyBank.setCurrentBalance(piggyBank.getCurrentBalance() - 
event.amount());
   }
   piggyBankProjectionPort .savePiggybank(piggyBank);

   return Optional.of(piggyBank);
}
```

this is projecting data
he said this is not hard.

Questions?