CQRS / EventSourcing

Softwerkskammer Leipzig

2016-10-18

Agenda

- CQRS in 10 mins
- EventSourcing in 10 mins
- Q&A
- Workshop intro
- Workshop (yes, **you** code) 1-3
- break
- Workshop (yes, **you** code) 4-7
- Q&A

Kudos

Oliver Wolf

https://innoq.com

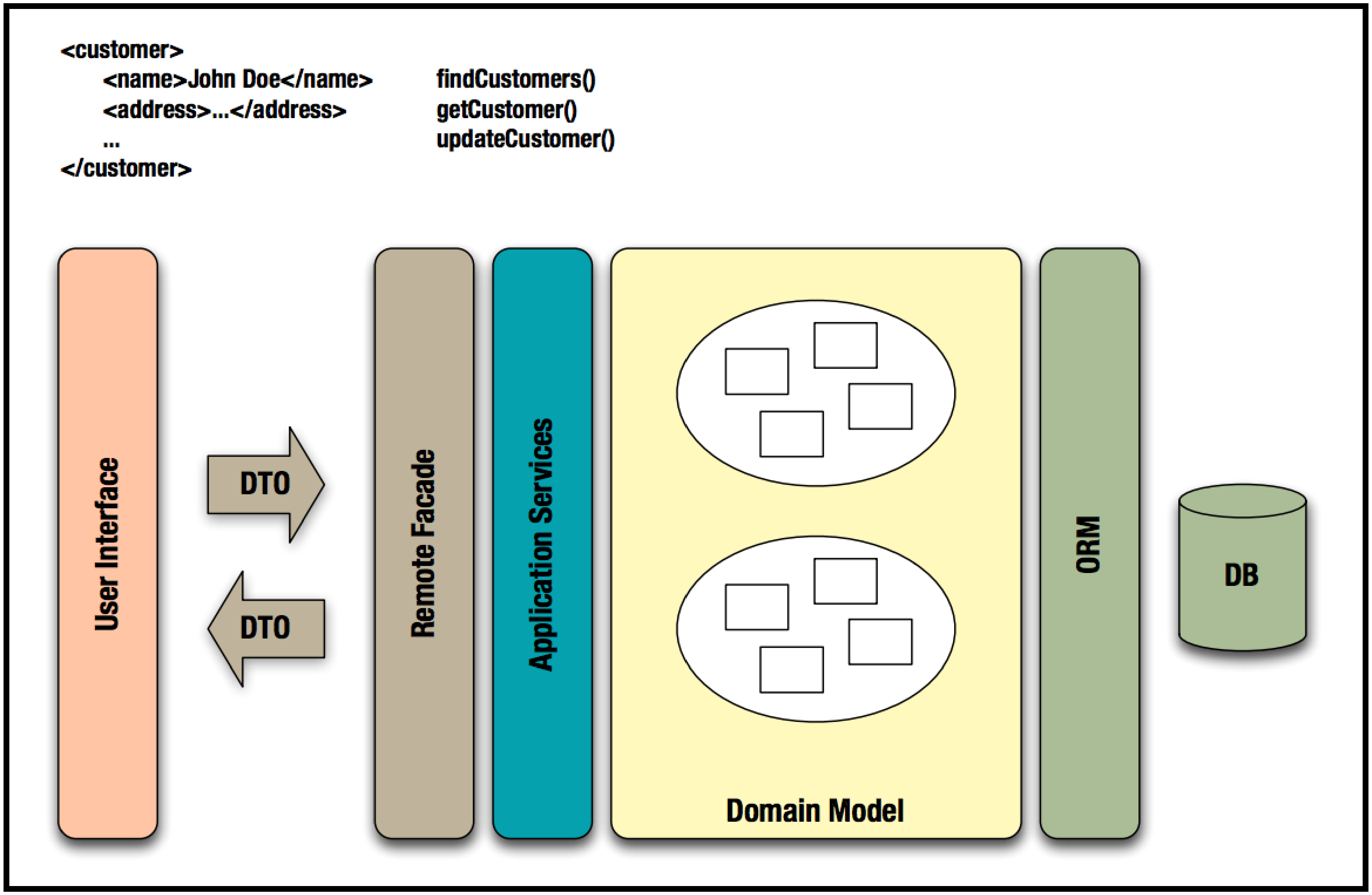
Greg Young

https://goodenoughsoftware.net/

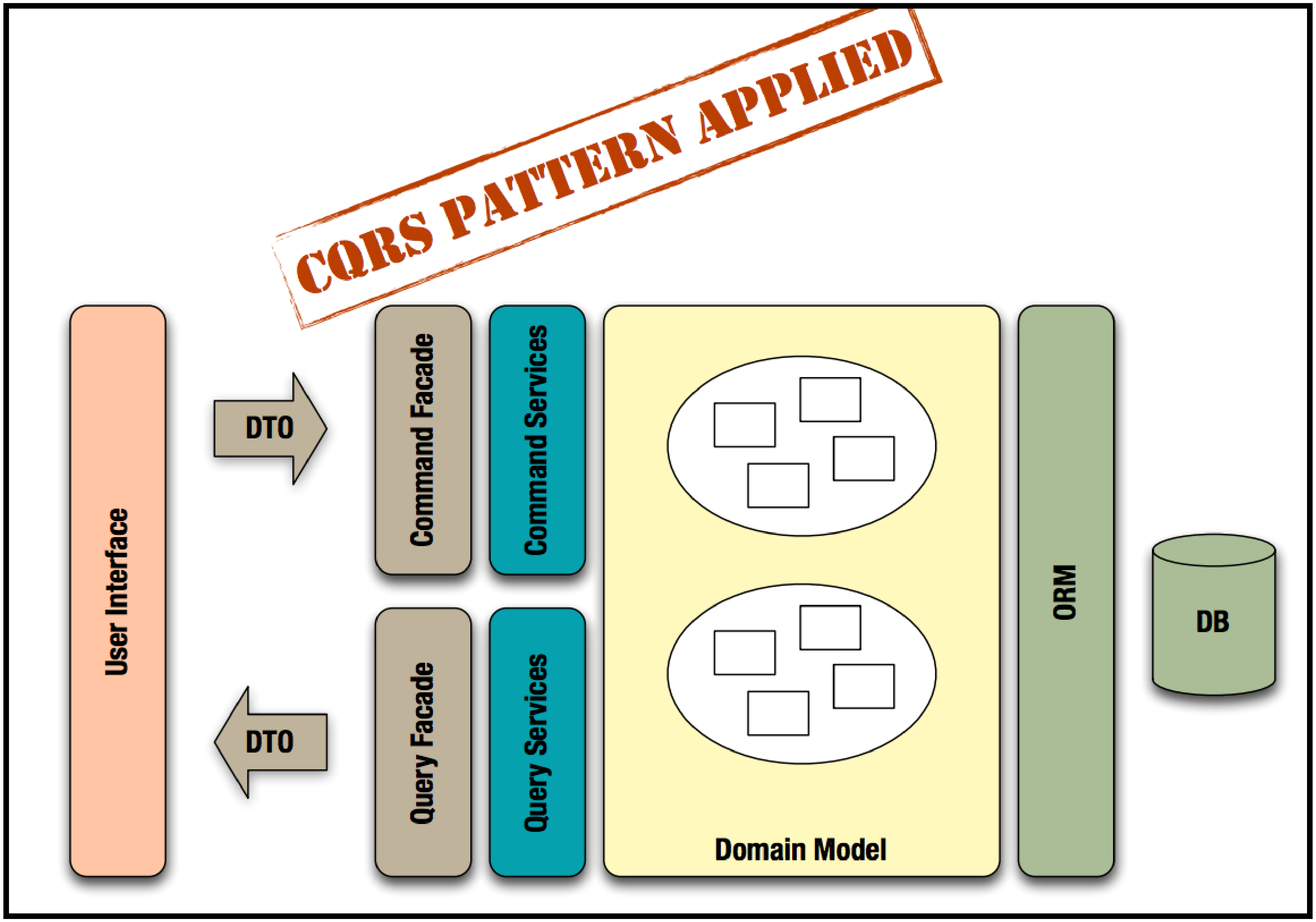
"You can do CQRS without EventSourcing, but you cannot do EventSourcing without CQRS."

CQRS

COMMAND QUERY RESPONSIBILITY SEGREGATION



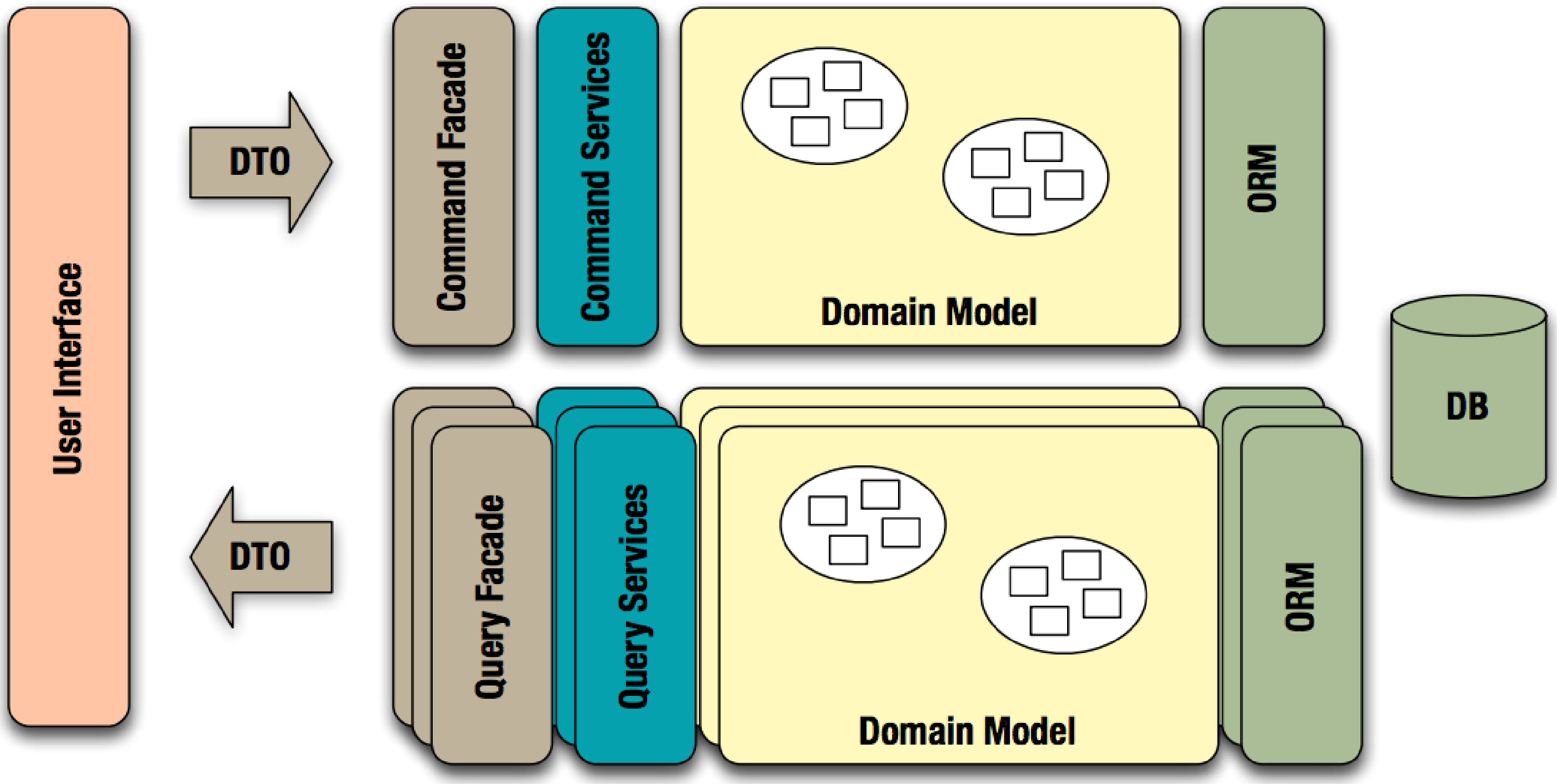
Traditional Layered Architecture



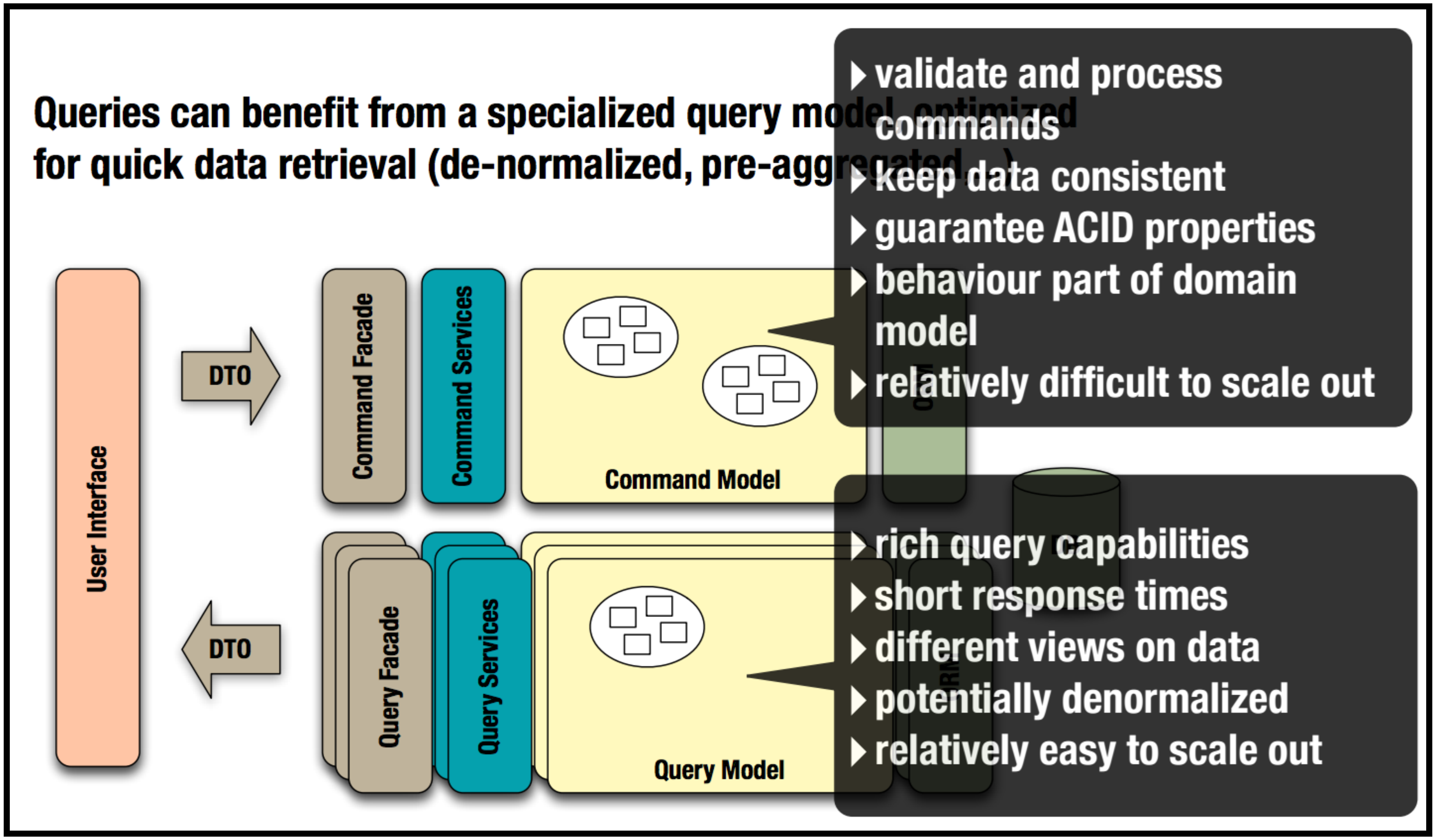
CQRS Pattern applied – done.

THAT IS IT!?

Command and query parts can scale independently, e.g. to accommodate highly asymmetric load.



pro: scale independently

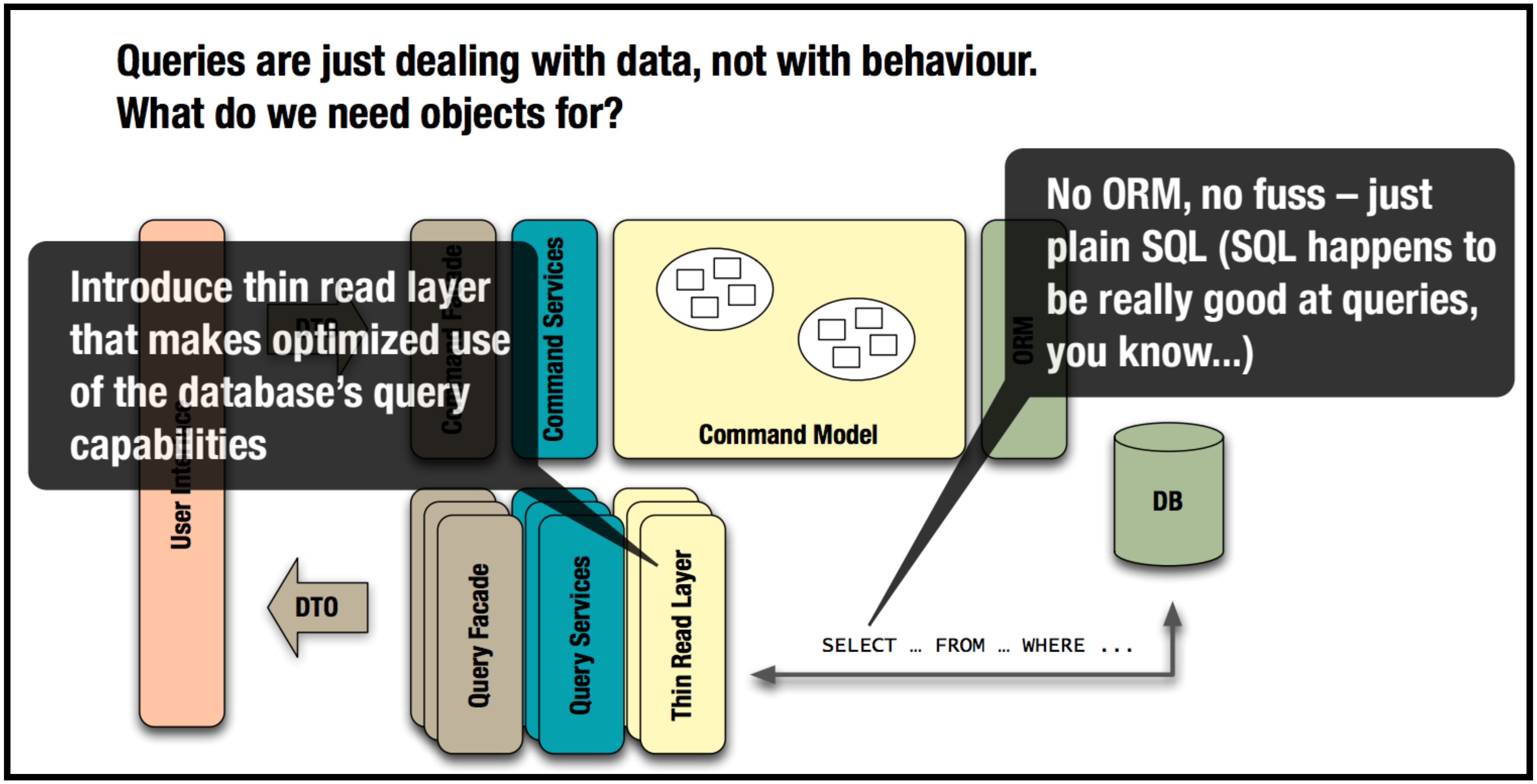


Queries can benefit from a specialized query model, optimized for quick data retrieval (de-normalized, pre-aggregated, ...)

- validate and process commands
- keep data consistent
- guarantee ACID properties
- behaviour part of domain model
- relatively difficult to scale out

- rich query capabilities
- short response times
- different views on data
- potentially denormalized
- relatively easy to scale out

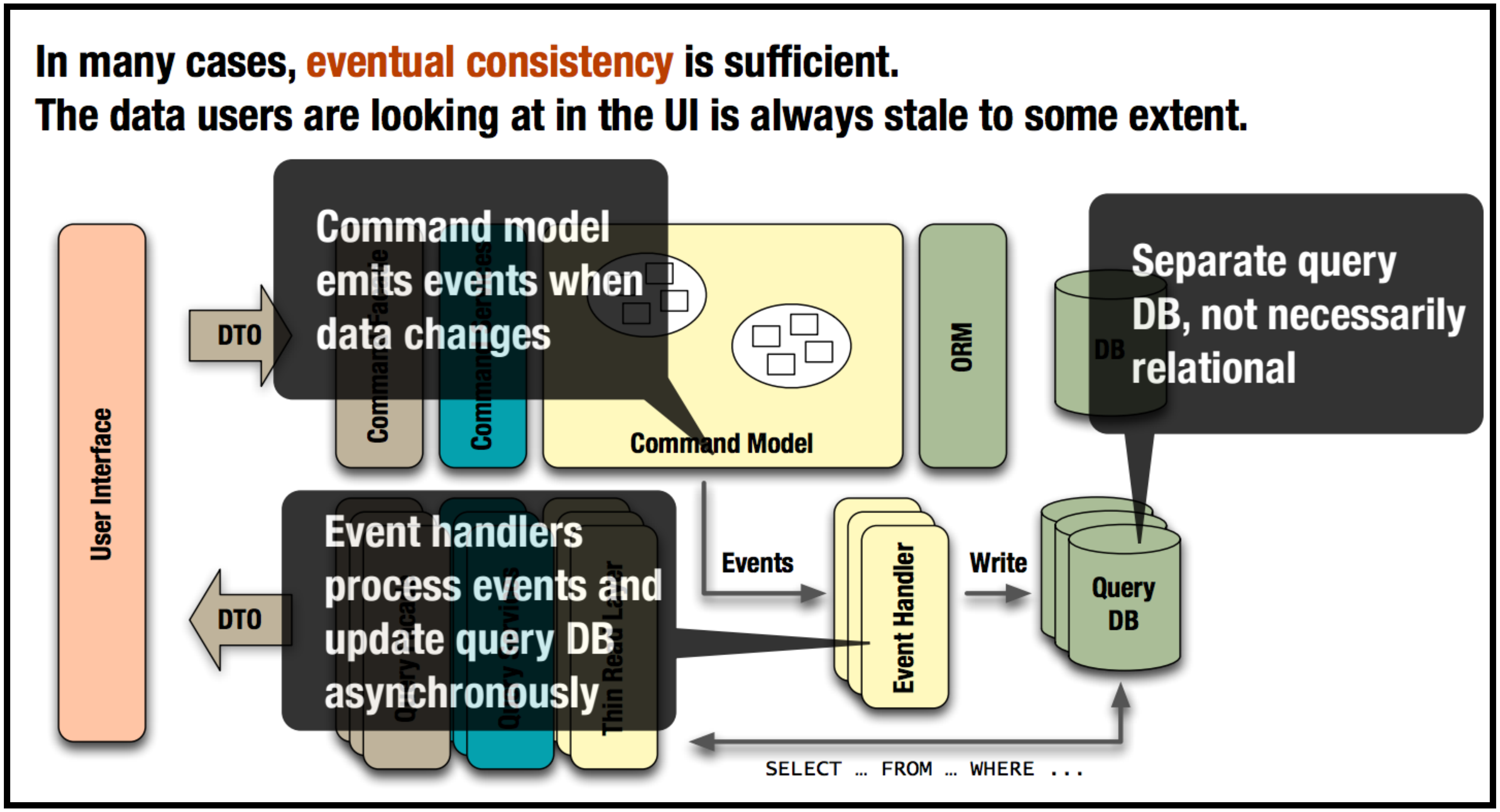
pro: optimized Query-Models (denormalization)


Queries are just dealing with data, not with behaviour. What do we need objects for?

Introduce thin read layer that makes optimized use of the database's query capabilities

No ORM, no fuss – just plain SQL (SQL happens to be really good at queries, you know...)

pro: **thin** read layer



In many cases, eventual consistency is sufficient.
The data users are looking at in the UI is always stale to some extent.

User Interface
DTO
Command model emits events when data changes
Command Model
ORM
DB
Separate query DB, not necessarily relational
Events
Event Handler
Write
Query DB
DTO
Event handlers process events and update query DB asynchronously
Thin Read Layer
SELECT … FROM … WHERE ...

pro: Eventual Consistency & Read-Replicas

Conclusion

CQRS helps with

- asymmetric load / read replicas
- gain from different QueryModels / Technologies
- helps Time-to-Market
- avoids technology Lock-In
- enables local optimization on Query-Models

EventSourcing

as a Concept

"An architectural pattern which warrants that your entities (as per Eric Evans' definition) do not track their internal state by means of *direct serialization or O/R mapping*, but by means of **reading and committing events** to an event store."

EventSourcing

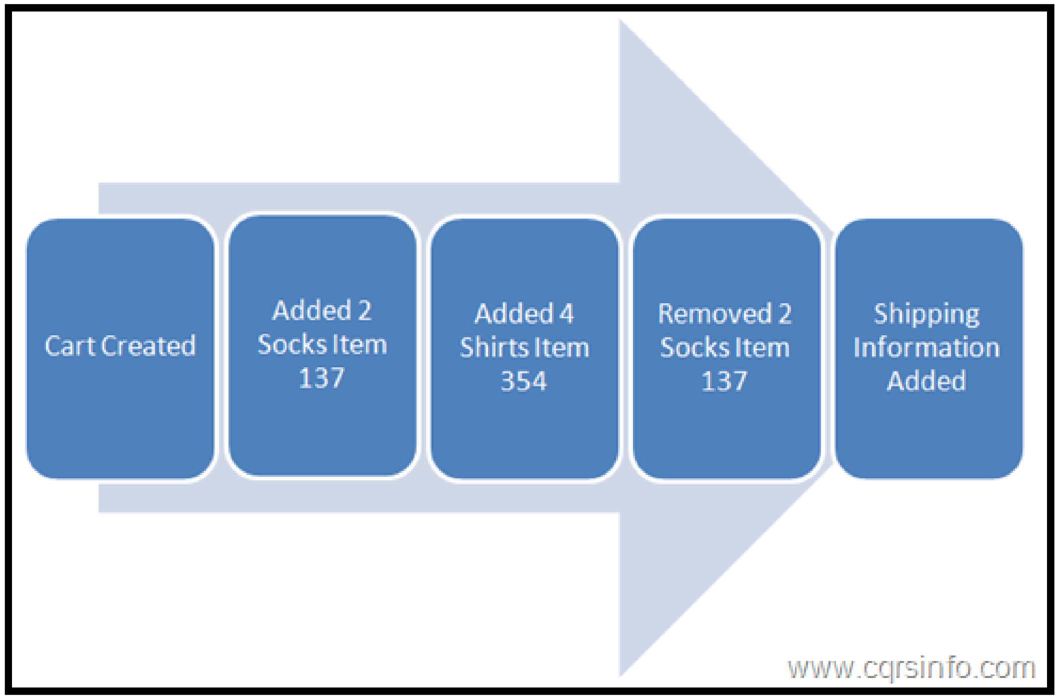
- Architectural Pattern
- EventStore keeps log of Events (Facts)
- 'current' State is
  - transient
  - disposable
  - fully/reliably reconstructible from series of Events

Current State is a Left Fold of Events λ

- FP: Left Fold aggregates a collection via a function and an initial value
  - Ruby:
    ```
    [1, 2, 3].inject(0, :+)             ==   6   # symbol fn name
    ```
  - Scala:
    ```
    List(1, 2, 3).foldLeft(0)(_ + _)    ==   6   // anon function
    ```

- Provide an initial state s_0 and a function f : (S, E) => S

- Current State after event e_3 is:
  - = leftFold([e_1, e_2, e_3], s_0, f)
  - = f(f(f(s_0, e_1), e_2), e_3)


Cart Created
Added 2 Socks Item 137
Added 4 Shirts Item 354
Removed 2 Socks Item 137
Shipping Information Added
www.cqrsinfo.com

Pros

- focus on **state transitions**, rather than data structure
- audit log already included
- reports over the past
- no Information-loss (Socks, Item 137)
- replayable
  - basically Time Machine (travel back and forth)
  - history of System state
  - helps debugging
  - no infamous SQL-migration-scripts, just change aggregation and replay

Cons

- maybe different angle to modeling
- (little) more complex than a CRUD System
- new Challenges like:
  - aggregation performance
  - evolving events
  - capacity

proven Patterns for the new challenges ***do*** exist!

Myths

- artificial approach to modeling (**not** true)
- requires eventual consistent (**not** true)
- inherently difficult & complex (**not** true)
- bad performance (**not** true)

Questions up to here?

EventSourcing Basics

Workshop

Disclaimer

the code here is

- **NOT** an EventSourcing framework !
- just for demonstration of concepts
- oversimplified
- Java, but can be done in any language
- very basic DI with Spring, but can be done without
- uses Lombok for brevity (just syntactical sugar)

There are many ways to skin a Cat

This Workshop is about

Discussion

not Code

EventSourcing Basics

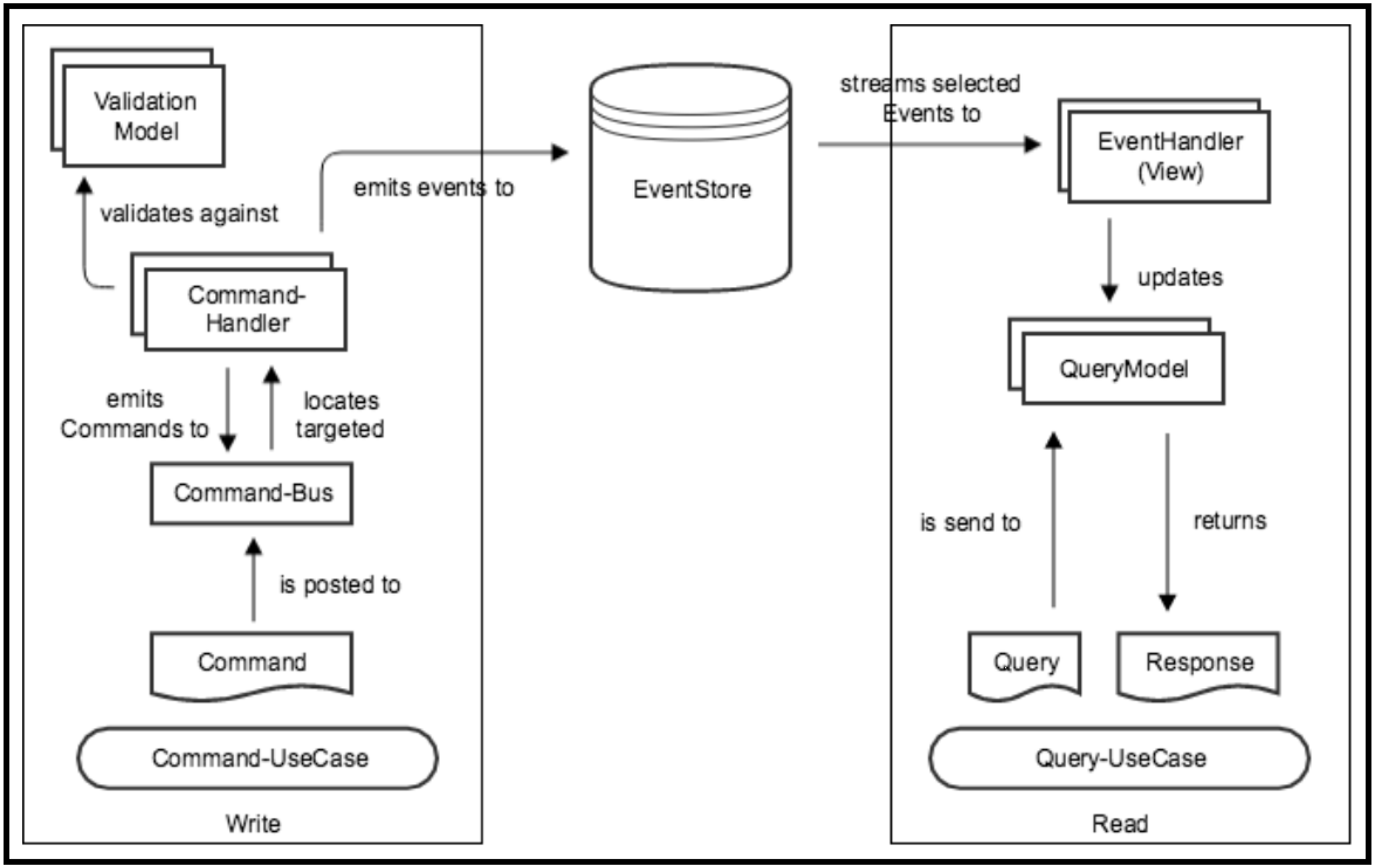
Intro

Hello, we are FooBank !

Our Domain

- Local Bank
- physical Counter
- will expand into online-banking

Infrastructure



Component **Responsibility**

Glossary

Component	**Responsibility**
ApplicationFacade	single Entry Point / internal API (optional)
Command	Request for the System to *do* something
CommandBus	find CmdHandler for given Cmd
CommandHandler	accept or reject Command, emit Effects
Effects	List of Messages
Message	Event or Command
Event	a given Fact
EventStore	a log of Events *that have happened*
EventHandler/View	process Events, project useful Model
Query	a Question to a Model
QueryModel	queried by the outside world, query-optimized data
ValidationModel	answers Queries while validating

EventSourcing Basics

Session 1

Aggregating to the canonical Domain Model

Canonical Domain Model

```java
public class Account {
    private final UUID id;
    private final String firstName;
    private final String lastName;
    private int balance = 0;

    void credit(int amount) { balance += amount; }
    void debit(int amount)  { balance -= amount; }
}
```

UseCase Deposition

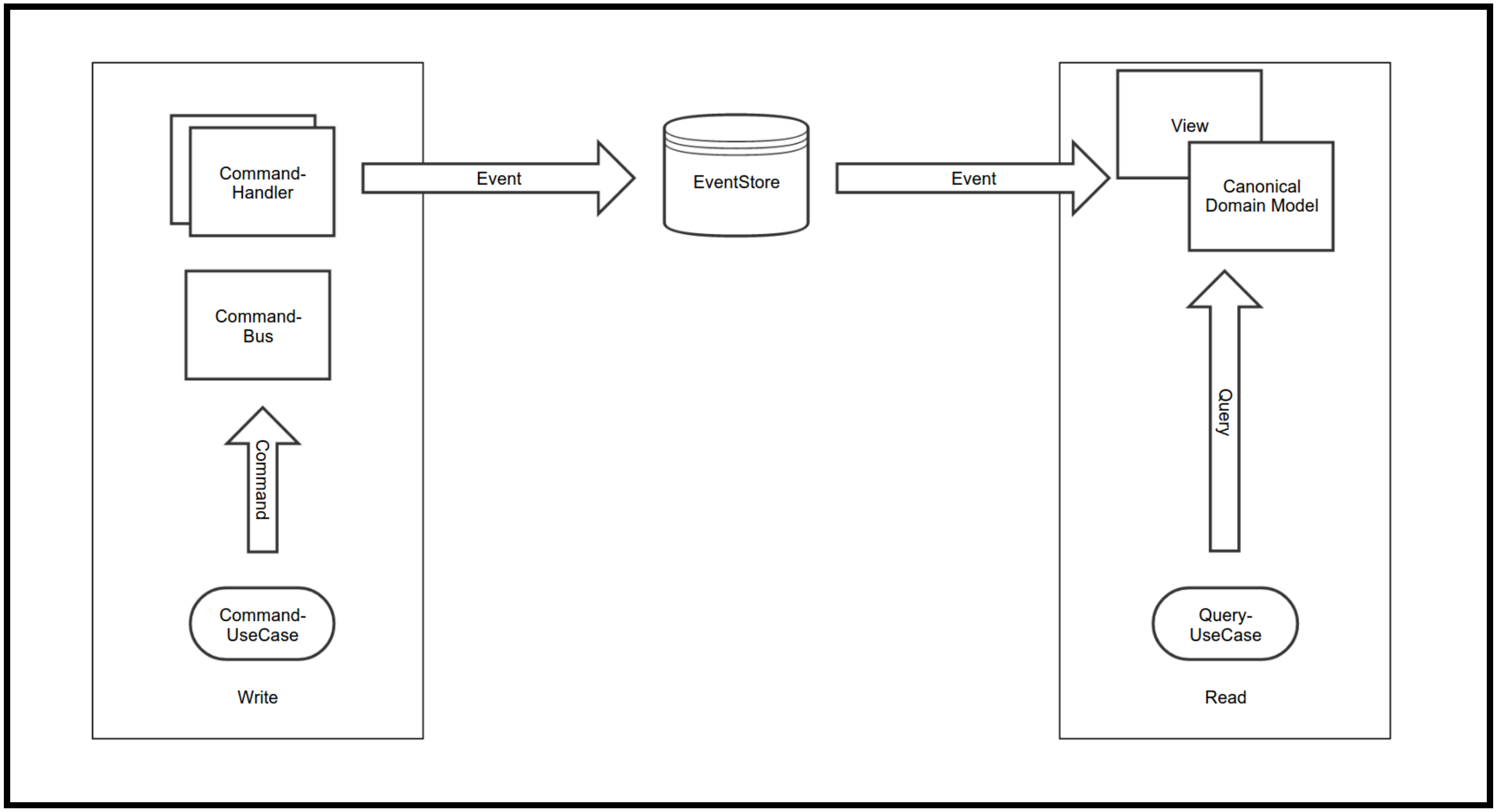
UseCase Deposition

As a Customer
i want to **deposit** cash at the counter
in order to credit it to my account.

UseCase Withdrawal

UseCase Withdrawal

As a Customer
i want to **withdraw** money from my account at the counter
in order to cash it out.


Command-Handler
Event
EventStore
Event
View
Canonical Domain Model
Command-Bus
Command
Query
Command-UseCase
Query-UseCase
Write
Read

What we learn

- Implement **write** side
  - minimal Commands
  - minimal CommandHandlers
  - minimal Events
- Implement **read** side
  - minimal EventHandlers (Views)
  - that populate the canonical Domain Model

```
git clone https://github.com/uweschaefer/es-basics.git
```

Session 1

1. Implement **ApplicationFacade.deposit/withdraw**
2. Create **Command classes** for both UseCases (see *CreateAccountCommand*)
3. Create **CommandHandlers** for both UseCases (see *CreateAccountHandler*)
4. Create **Event** classes for every UseCase (see *AccountCreatedEvent*)
5. Extend **AccountView** to aggregate Accounts
6. Pass the Tests

What just happened?

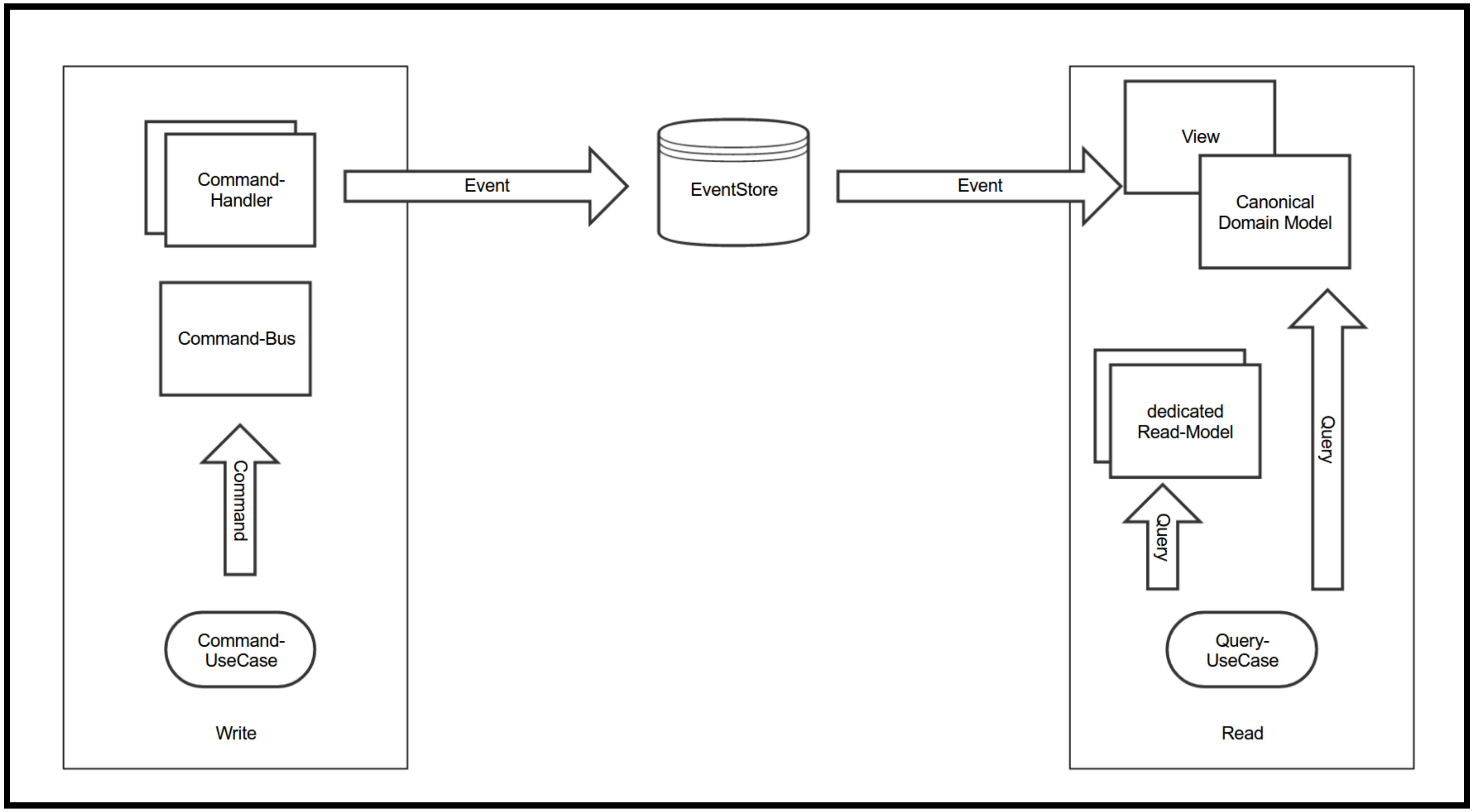
- complete Roundtrip
  - Command -> CommandHandler
  - Event -> EventHandler (View)
  - Query via DomainModel

EventSourcing Basics

Session 2

Dedicated QueryModel

Overview


Command-Handler
Command-Bus
Command
Command-UseCase
Write
Event
EventStore
Event
View
Canonical Domain Model
dedicated Read-Model
Query
Query
Query-UseCase
Read

Aggregation

AccountView = EventHandler that aggregates Events to an Account.

Not every Query within one Account

UseCase ValuedCustomerReport

UseCase ValuedCustomerReport

As a Manager, i want
a complete report the lists all *valued customers*
in order to free them from handling charges.

Specification ValuedCustomer

Specification ValuedCustomer

a *valued Customer*
has deposited an amount of **>=1000€ at least twice**.

How **not** to do that

Iterate Accounts and inspect their Depositions one by one.

What we learn

- Aggregate beyond Entity-Boundaries
- Create **dedicated Read Model**
  - find appropriate DataStructure
  - select Event-Types by @EventConsumer Methods

Session 2

1. git clean -fd && git reset --hard session2
2. implement ValuedCustomerReportView
3. pass the Tests

What just happened?

- we added a dedicated Read/Query-Model
  - beyond aggregate boundaries
  - Query-optimized Datastructure
  - PullViews have to actually **pull** the events from the ES at some point.

EventSourcing Basics

Session 3 (Bonuslevel)

Rolling Snapshot QueryModel

Overview

- ValuedCustomerSupport aggregates ALL depositions in the System
- gets slower & slower
- how to tacle that?

Problematic

every time a Report is needed, a new View has to be created.

```
facade.deposit(...);

ValuedCustomerReportView report1 = new ValuedCustomerReportView(es);
assertTrue(report1.isValuedCustomer(...));

facade.deposit(...);

ValuedCustomerReportView report2 = new ValuedCustomerReportView(es);
assertTrue(report2.isValuedCustomer(...));
```

What if?

we could reuse a QueryModel, that is being updated, rather than re-created?

```
ValuedCustomerReportView report = new ValuedCustomerReportView(es);

facade.deposit(...);
assertTrue(report.isValuedCustomer(...));

facade.deposit(...);
assertTrue(report.isValuedCustomer(...));
```

---

discuss what would be necessary, conceptually?

No, really – **Discuss**

What we just learned?

- Concept of Rolling Snapshot

Session 3

1. git clean -fd && git reset --hard session3
2. look at *View.last*, *View.apply* and *PullView.pullEvents*
3. change ValueCustomerReport appropriately.
4. pass the Tests

What just happened?

- Rolling Snapshot
  - keeps latest
- Eventstore provides query of EventStream from **after** a particular event
- And yes, we can have more snapshots than one, if needed

Have a break.

Have a KitKat

EventSourcing Basics

Session 4

Event Design

UseCase Transfer

UseCase Transfer

As a user, i want to
transfer Money from my account to someone else's
in order to pay my rent online.

Acceptance Criteria

- *AccountUnknownException* if receiver or sender account does not exist
- *UnfundedTransferException* if sender does not have enough money (no debt allowed)

What we learn

- Event granularity matters
- Events need to reveal their intent
- Use of a ValidationModel
- Commands can be rejected

Session 4

1. git clean -fd && git reset --hard session4
2. Implement *TransferHandler*
3. pass the Tests

What just happened?

- Granularity: Events belong to ONE Aggregate
  - we need SendTransfer, RecieveTransfer
- Events reveal intent
  - do not reuse WithdrawnEvent etc, its a different UseCase!

EventSourcing Basics

Session 5 (Bonuslevel)

Dedicated WriteModel / ValidationModel

Command validation sometimes needs Context

Your take on Criteria 1?

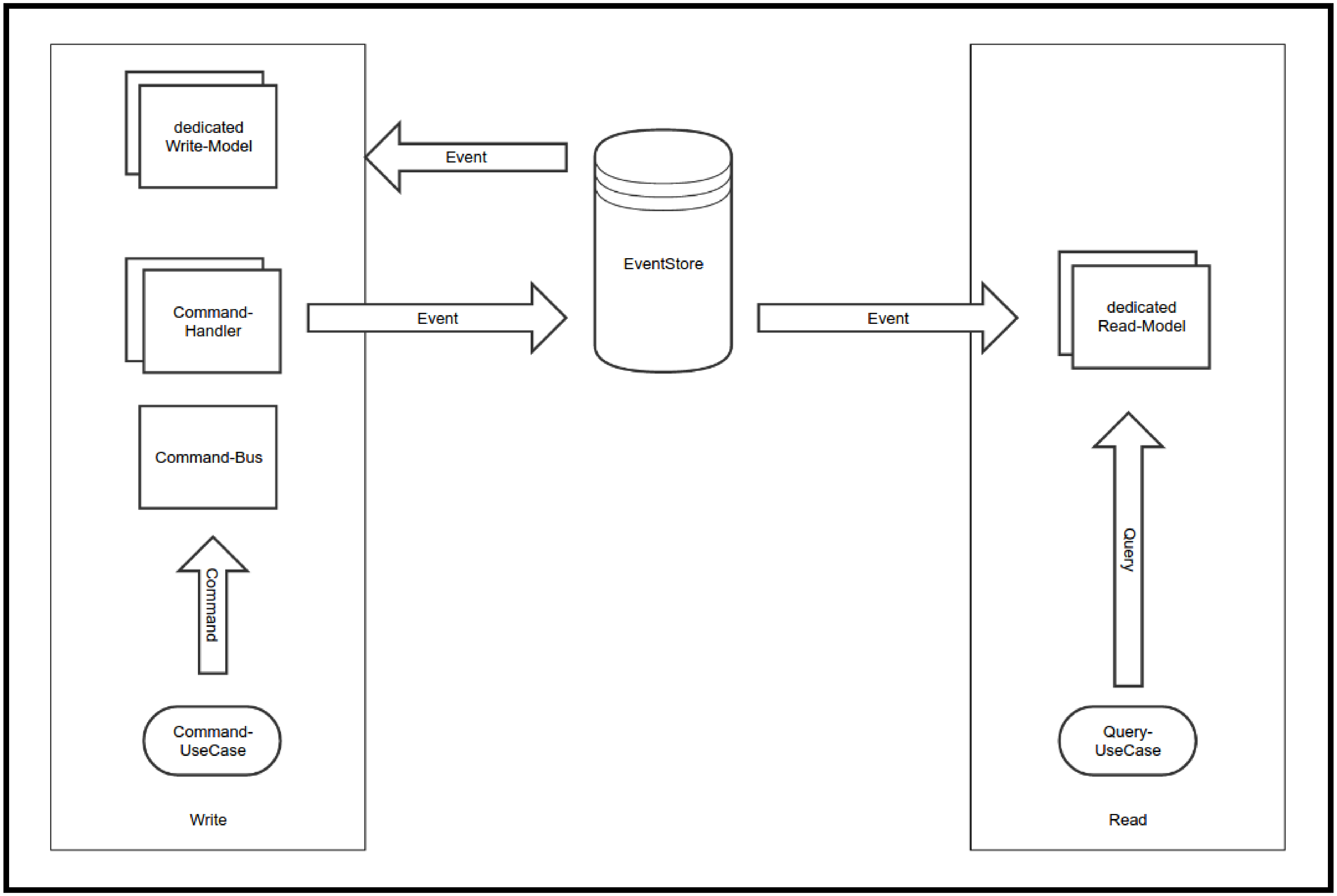
- *AccountUnknownException* if receiver or sender account does not exist

```
private boolean exists(UUID id) {
    return repo.find(id) != null;
}
```

Using AccountView just to find out, if an Account exists is wasteful.

All we really need to know is if the aggregate exists.

Overview


dedicated Write-Model
Event
Command-Handler
Event
EventStore
Event
dedicated Read-Model
Command-Bus
Command
Query
Command-UseCase
Query-UseCase
Write
Read

CommandHandler

Responsibilities

- validate command
- accept or reject command based on that validation
- emit Messages on accepting

Session 5

1. git clean -fd && git reset --hard session5
2. implement KnownAccountsView
3. pass the Tests

What just happened?

- dedicated WriteModel / ValidationModel
- does not have to be DomainModel, as it does not need behaviour

EventSourcing Basics

Session 6

Side-Effects

- Some Commands may trigger external behavior.
- Replaying that would be problematic.

UseCase Notification

UseCase Notification

As a user, i want to
be notified by email when i recieve a transfer
in order to buy champagne asap.

What we learn

- how/where to model Side-Effect

Session 6

1. git clean -fd && git reset --hard session6
2. use CreditNotificationService to send mail
3. discuss where/how to do it properly
4. hint: see CommandBus.publish()
5. pass the Tests

What just happened?

- CommandBus has to be reliable
- Commands can be Effects, too
- Side-Effects can be modeled as Commands / CommandHandlers

Intermission

What about Consistency?

Did we relax Consistency compared to a normal CRUD/ORM implementation?

NO!

But where we could, how can we take advantage?

EventSourcing Basics

Session 7 (Bonuslevel)

Push-Views

Up to now, all views have been *PullView*s, that call *pullEvents()* to stream events into them.

Pro

- we can define when to update the View's State

Con

- we have to Query the EventStore in order to know, if View's State is stale
  - the more Queries we run, the more catastrophic this is:
  - bad Latency for Queries
  - high Contention on EventStore

UseCase GoldCustomers

UseCase GoldCustomers

As an accountant, i want to
know all the Gold-Customers
in order to be extra nice to them.

Specification Gold Customer

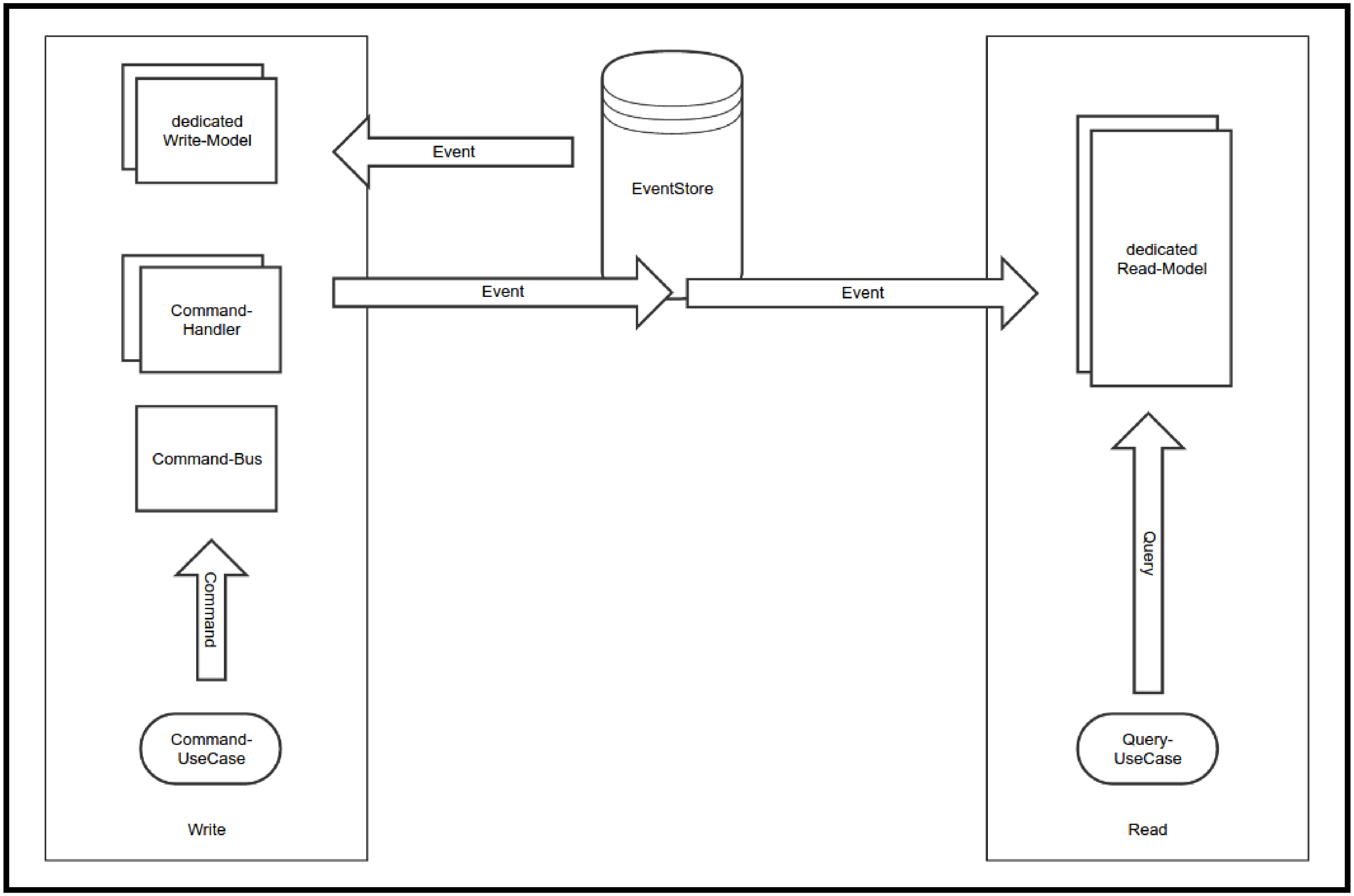
Specification Gold Customer

someone who recieved a transfer **>=10.000€**
at least once

Acceptance Criteria

- report must be **instant!** (low-latency)
- report must be a collection of Strings "<LASTNAME>, <FIRSTNAME>"
- order is not important
- only Transfers count – Depositions **must not** be examined
- report does not need to include GoldCustomer that recieved the status in the last few seconds...

... which means **Eventual Consistency** is ok


dedicated Write-Model
Event
EventStore
Command-Handler
Event
Event
dedicated Read-Model
Command-Bus
Command
Query
Command-UseCase
Query-UseCase
Write
Read

What would be necessary to push events to the view?

What we learn

- Use Push-Model for Views
- pros and cons of push vs pull

Session 7

1. git clean -fd && git reset --hard session7
2. implement GoldCustomers *extends PushView*
3. pass the Tests

What just happened?

- Implemented a push-View that is updated by processing Events asynchronously
- Push reduces read latency
- introduces eventual consistency
- introduces concurrency
- PushViews mostly unusable as Validation Model (not strictly consistent)

One possible solution can be found here

```
git clean -fd && git reset --hard theend
```

Links

- O.Wolfs CQRS Slides
  - https://speakerdeck.com/owolf/cqrs-for-great-good-2
- Greg Young's Blog
  - https://goodenoughsoftware.net/
- Axon mature ES Framework
  - http://www.axonframework.org/
- Lagom Modern ES Framework based on Akka
  - https://www.lightbend.com/lagom
- Microsofts CQRS/ES Patterns & Practices
  - https://msdn.microsoft.com/en-us/library/jj554200.aspx

Q & A