dutchworks™

Practical CQRS

Seven League Boots or just a fairy tale?

Allard Buijze @ Goto Amsterdam 2011

Allard Buijze

- Software Architect at Dutchworks
  - formerly known as JTeam
- 10 years of web development experience
- Strong believer in DDD and CQRS
- Developer and initiator of Axon Framework
  - CQRS Framework for Java
  - www.axonframework.org

Viewer advisory

This is a true story, but some slides have been dramatized. Names have been changed to protect the innocent.

Contains nerdy language

Once upon a time, in a country far far away

There was a great man, with big blue boots

They are able to tackle the ultimate evil: Complexity

Everyone wanted them…

But one man took them further…

He said: “Use one for commands, and one for queries…”

"… and free the world of Complexity. Forever!!"

IT lived happily ever after…

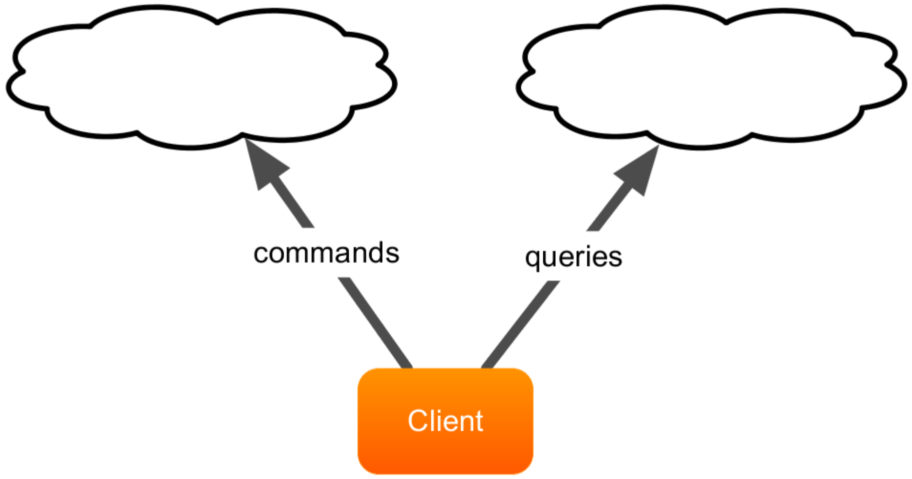
The End

But will it work in my world?

- Deadlines, Pressure
- Changing requirements, Renewed insights
- Performance
- Team experience, Learning curve

CQRS – A brief introduction

- Separation of components
  - Command Handling
  - Execution of queries



- Why?
  - Non-functional requirements
  - Concurrency and staleness
  - Domain model complexity

dutchworks™

Non functional requirements

- Response time requirements
  - Google search: < 100ms
  - Credit card payment: 10 seconds

- Command to query ratio
  - 1 to 10 ?
  - 1 to 100 ?

Concurrency and staleness

- Multiple users acting on the same data

- Decisions are based on stale data

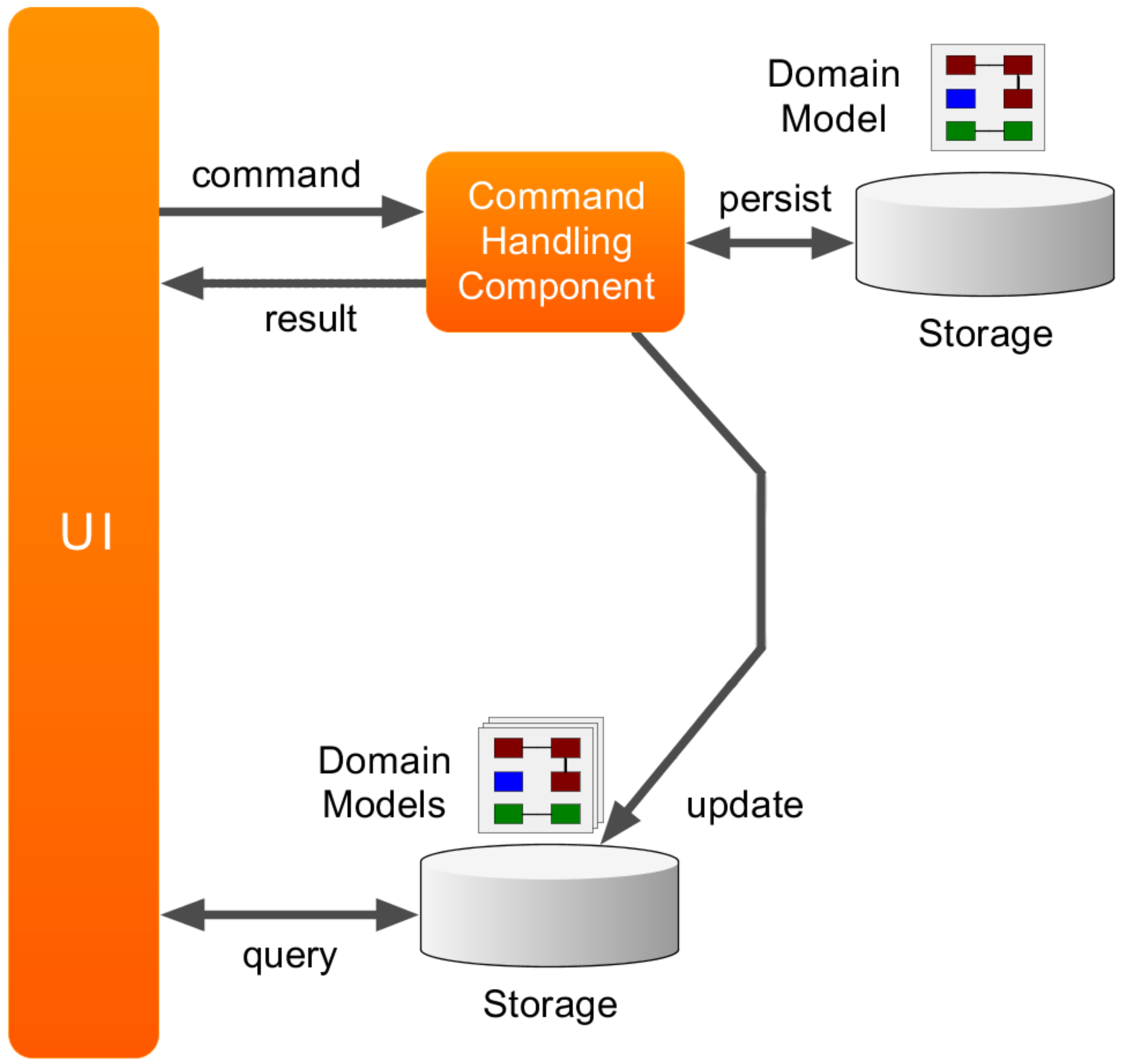
Domain model

- Simplified representation of concepts in a domain to solve specific problems

- Applications solve 2 types of problems:
  - Change state
  - Expose state

- CQRS: Create a domain model for each purpose

CQRS Overview

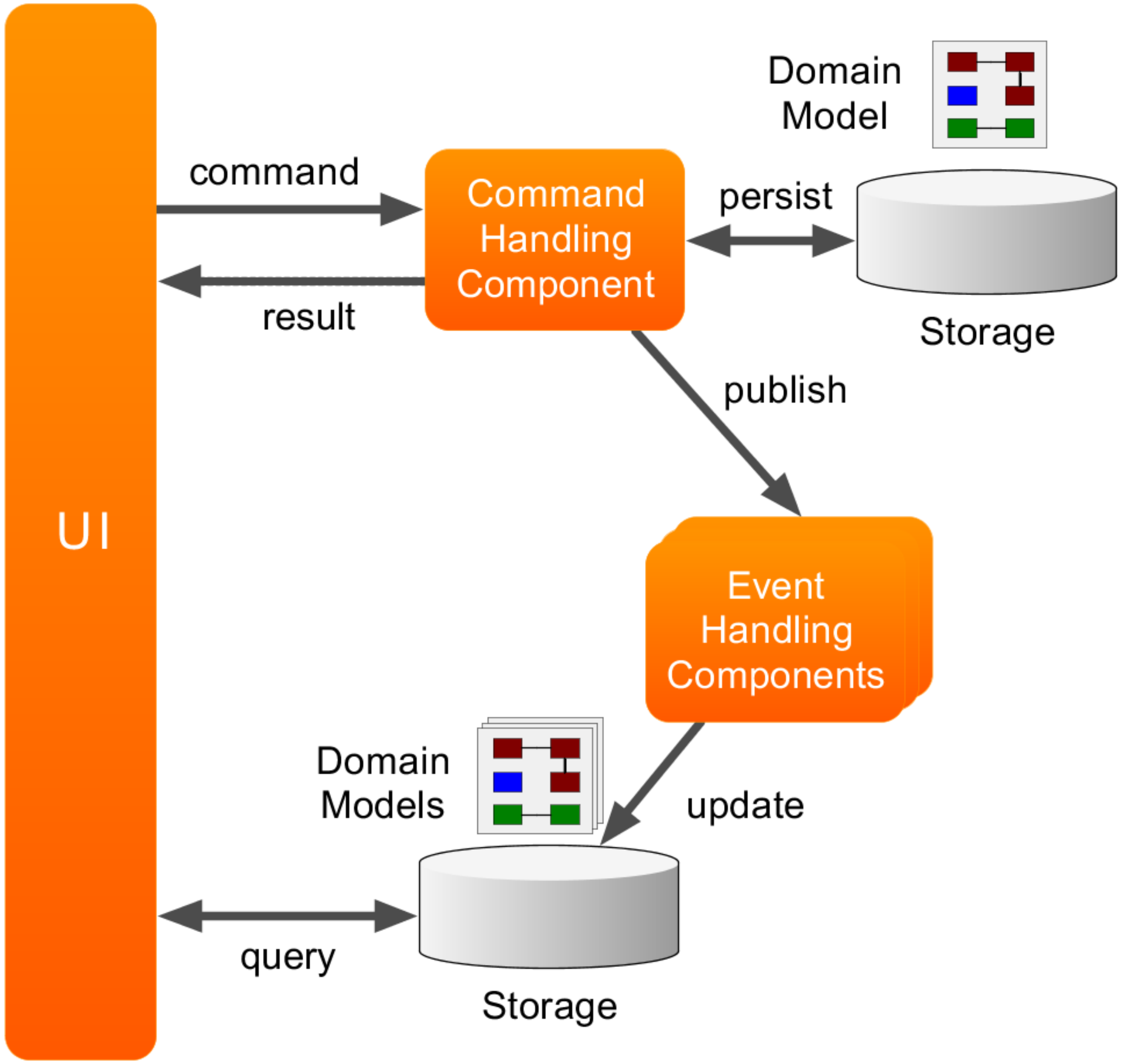


CQRS supports scalability

- Embrace staleness

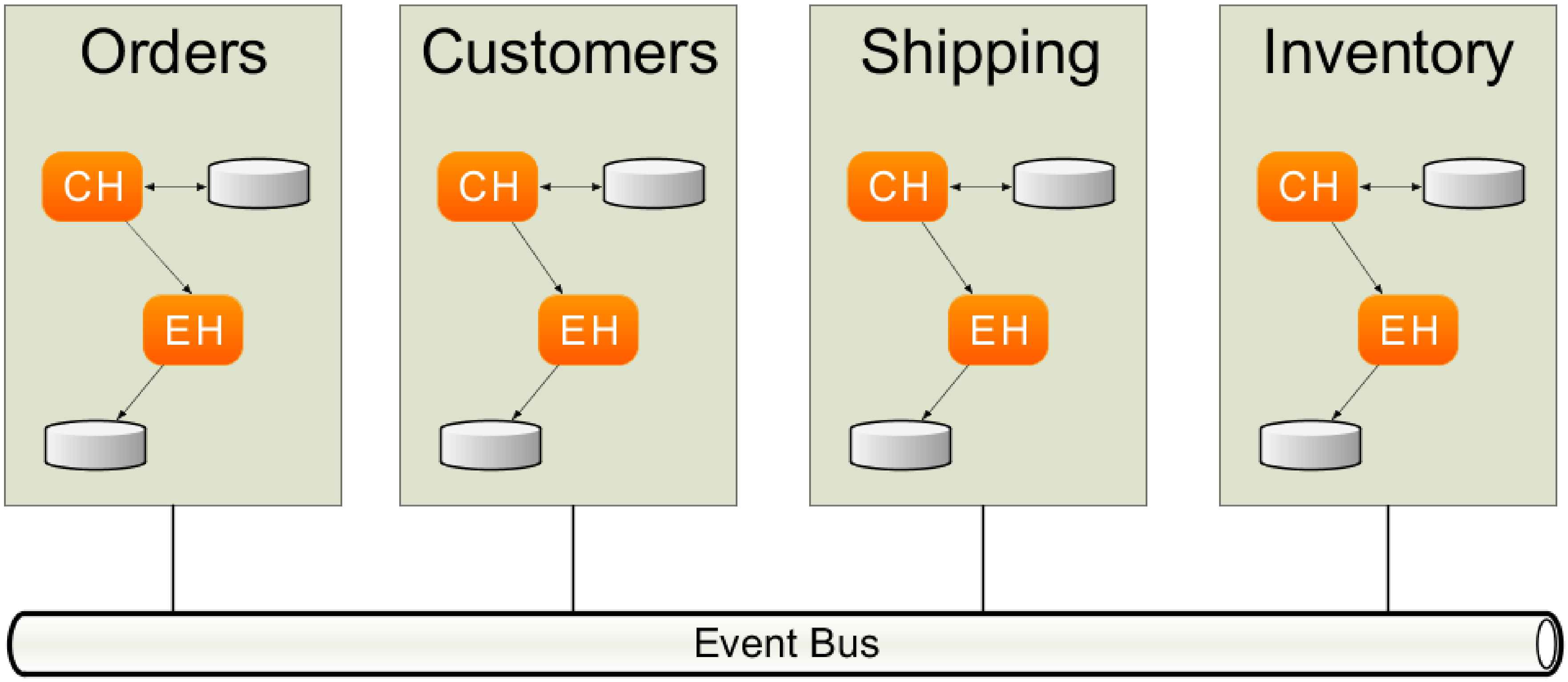
- And get: scalability

CQRS + EDA Overview



dutchworks™

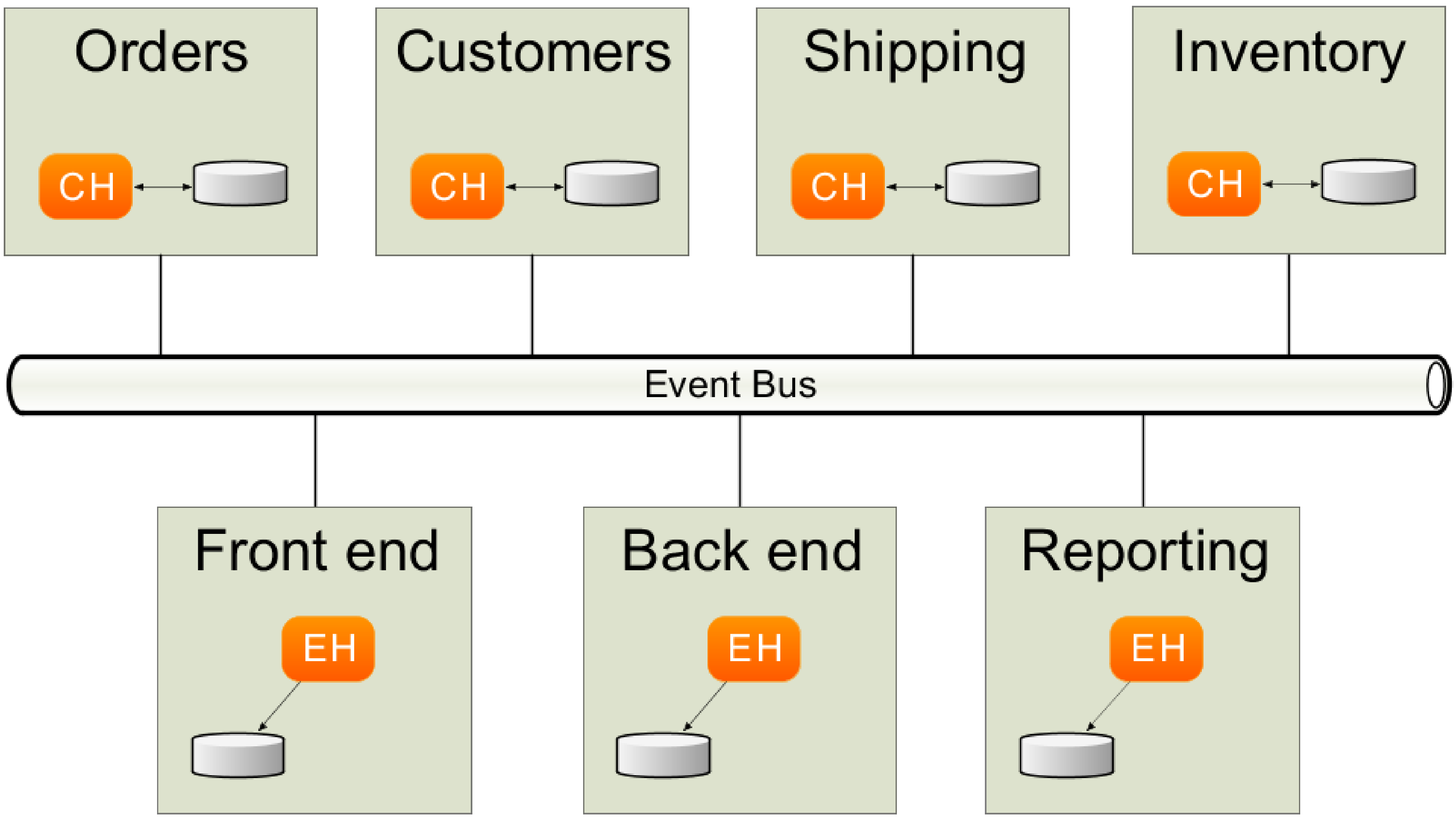
Scalability



CH Command Handler   EH Event Handler

dutchworks™

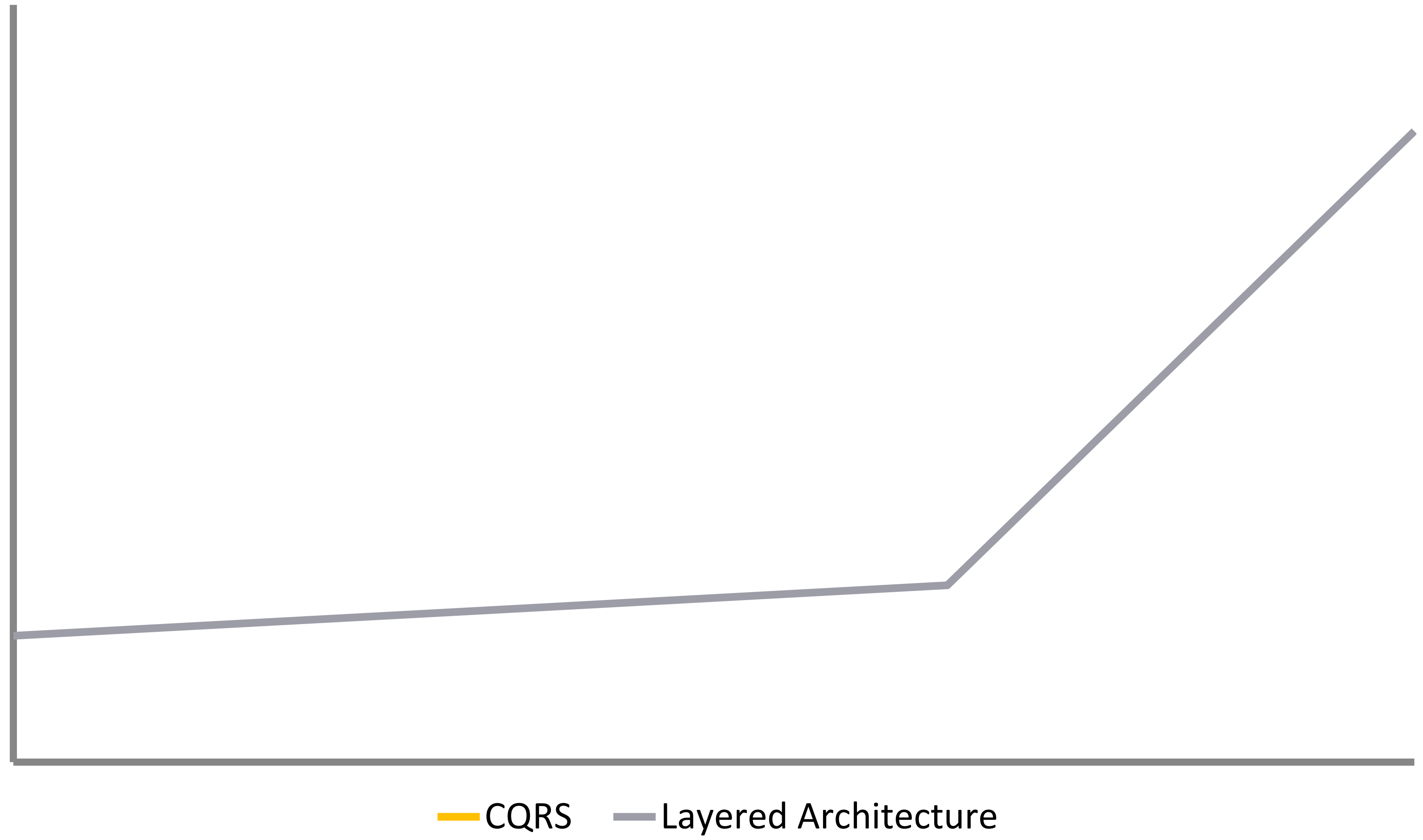
Scalability



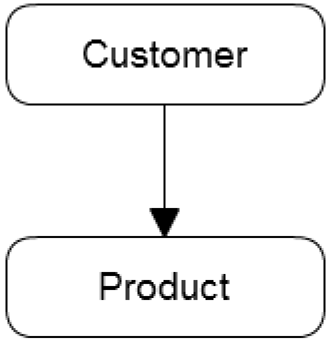
CQRS in our world

- Scalability is barely an issue for most applications
- Complexity is what hits most of them!

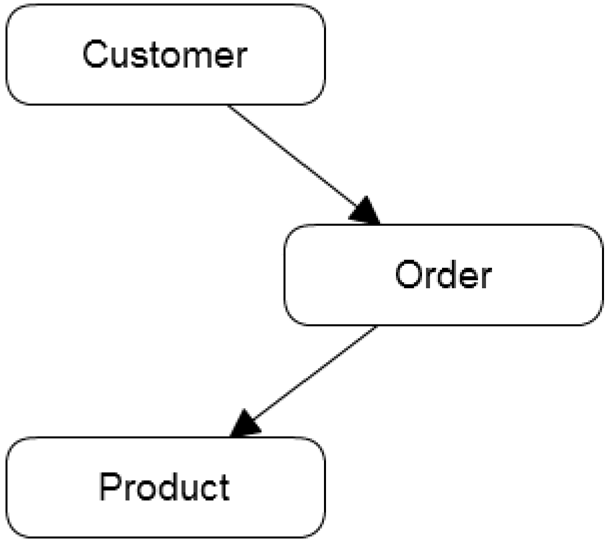
Evolution of complexity



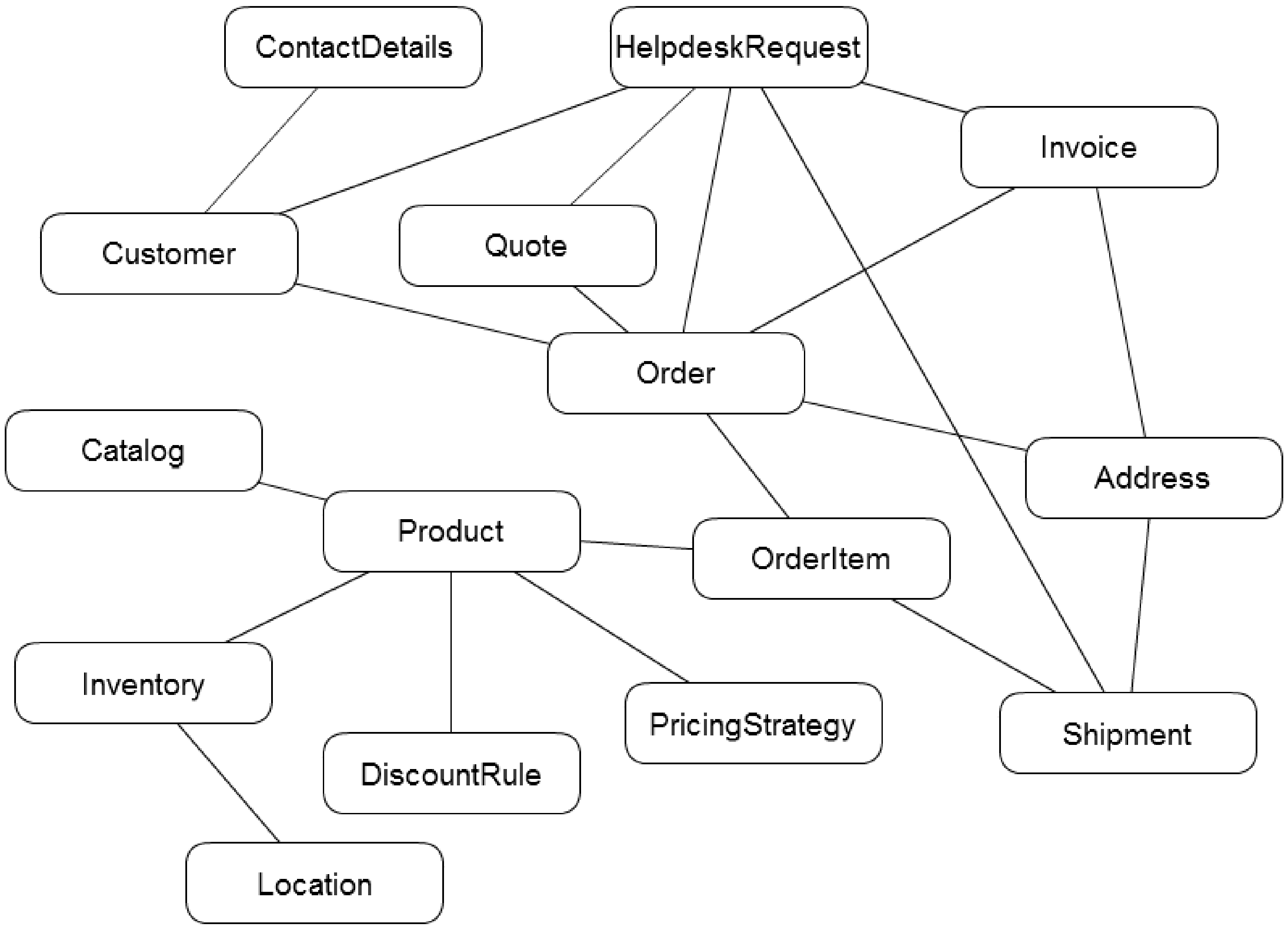
Evolution of a domain model



Evolution of a domain model



Evolution of a domain model



Complexity…

```
private static final String PLAYER_COCKPIT_WATERFALL_ITEMS_QUERY =
        "(" +
          "select id, " + EntityType.NEWS_ITEM.ordinal() + " as entity_type, publish_date as sort_date " +
          "from news_item " +
          "where active = true and (" +
              "poster_player_id = :playerId " +
              "or poster_player_id in (" +
                "select destination_friend_id from friendship where origin_friend_id = :playerId " +
              ") " +
              "or project_id in (" +
                "select distinct project_id " +
                "from donation " +
                "where donor_participant_id = :playerId and status = 'OK'" +
              ")" +
              "or project_id in (" +
                "select project_id from ambassador_project where player_id = :playerId " +
              "))" +
        ") union all (" +
          "select id, " + EntityType.DONATION.ordinal() + " as entity_type, approval_date as sort_date " +
          "from donation " +
          "where status = 'OK' and (" +
              "donor_participant_id = :playerId " +
              "or donor_participant_id in (" +
                "select destination_friend_id from friendship where origin_friend_id = :playerId" +
              ")" +
              "or raised_via_player_id = :playerId " +
              "or raised_via_player_id in (" +
                "select destination_friend_id from friendship where origin_friend_id = :playerId" +
              ") " +
        ") union all ( " +
          "select id, " + EntityType.FRIENDSHIP.ordinal() + " as entity_type, created as sort_date " +
          "from friendship " +
          "where origin_friend_id = :playerId or (origin_friend_id in ( " +
             "select destination_friend_id from friendship where origin_friend_id = :playerId " +
          ") and destination_friend_id <> :playerId)" +
        ") ";
```

SELECT

NEWS_ITEM

or project in (…

status = ‘OK’

or project in (…

UNION ALL

DONATION

status = ‘OK’

or raised_via_player in (…

UNION ALL

FRIENDSHIP

CQRS and complexity

- Clear bounded contexts
- Decoupling between components
- No SQL “join-join-join” hell
- Clear definition of “core API”
  - In: Commands
  - Out: Events

Models in CQRS

- Command Model
  - “Core-API”
  - Driven by behavior

- Query
  - Table-per-view
  - Driven by data needs

Command model

- Only store information that influences a command’s outcome (i.e. behavior)
- Built up of aggregates (consistency boundaries)

Order date	✗
Order status	✓
Order amount	✓✗
Order description	✗

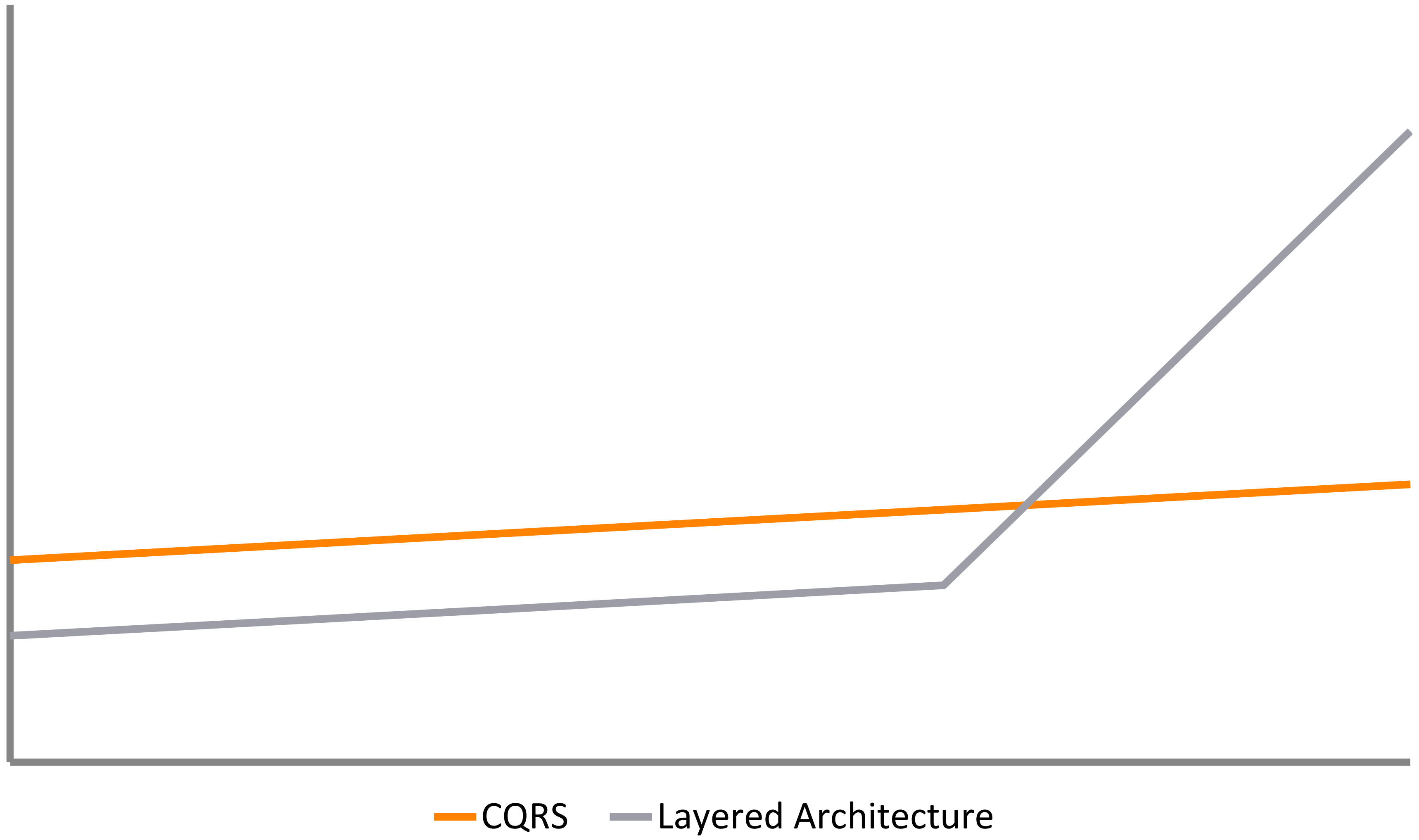
dutchworks™

Query model

- Stores what you want to see, the way you want to see it
  - Table per view
  - Persistent view model

- Watch your normalization!
  - Don’t over-normalize
  - Must fit UI information need

Evolution of complexity



CQRS applied – In a project

- Project: On-line Bridge platform

- Challenges & Requirements:
  - Scalability, Extensibility
  - “Perceived performance”, real-time feedback
  - Fraud prevention/detection

- Tools & Frameworks:
  - Java, Google Web Toolkit, Spring Framework, Axon Framework

Axon Framework

- Java
- Provides building blocks for CQRS applications
  - Event Bus, Command Bus, annotation based handlers
  - Support for Event Sourcing
  - Sagas
  - Given-when-then test fixtures
- Current version: 1.2
- More information: AxonFramework.org

Application components

Front-end
- Display game state
- Catch user actions

Game engine
- Keep track of game state
- Enforces Bridge rules
- Process commands

Tournament engine
- Game coordination
- Player ranking
- Process commands

Query component
- Pushes events to clients
- Executes queries

Event Store
- Stores events
- Source of engine state

Bounded Contexts

- Game and Tournament
  - Clearly separated
  - Each has a separate “core API”

- Improves maintainability

- Easy to implement new tournament types

- Contexts are “synchronized” using Sagas

Event Sourcing

- Storage option for command model
- Past events contain invaluable data
- Fraud detection a posteriori
- Build new features
  - Concept of “Credits” was added later
  - Management reports based data from day 1
- Gameplay analysis

Scalability

- Scaling out is straightforward
  - No need to change architectural features
  - No need to change application logic
- Step 1: Each context on a different machine
  - Publish events over a message broker (e.g. RabbitMQ)
- Step 2: Duplicate a context
  - Route commands based on targeted aggregate identifier
  - Consistent hashing

dutchworks™

Only for Bridge?

Types of projects using Axon Framework

- Electronic Medical Record
- License management for e-learning
- Pension value calculations
- Surgical tool tracking
- ...

Only blue skies & puffy clouds?

- Modeling not always easy
  - Modeling skills are absolutely required
  - Don’t be afraid to change your model

- Event Sourcing
  - Takes getting used to
  - Makes aggregate boundaries very strict
  - Requires developer discipline
  - Event Sourcing makes model changes a bit harder

Conclusion

- CQRS is a very simple architectural pattern
- When using events, it allows for easy scalability and extensibility
- Has a learning curve, but ROI is fast
- A good tool in the toolbox
- More “Seven League Boots” than “Fairy tale”!

Thank you

More information:

- **CqrsInfo.com**
- **DomainDrivenDesign.org**
- **AxonFramework.org**

Don’t forget to vote!

🟩 🟨 🟥