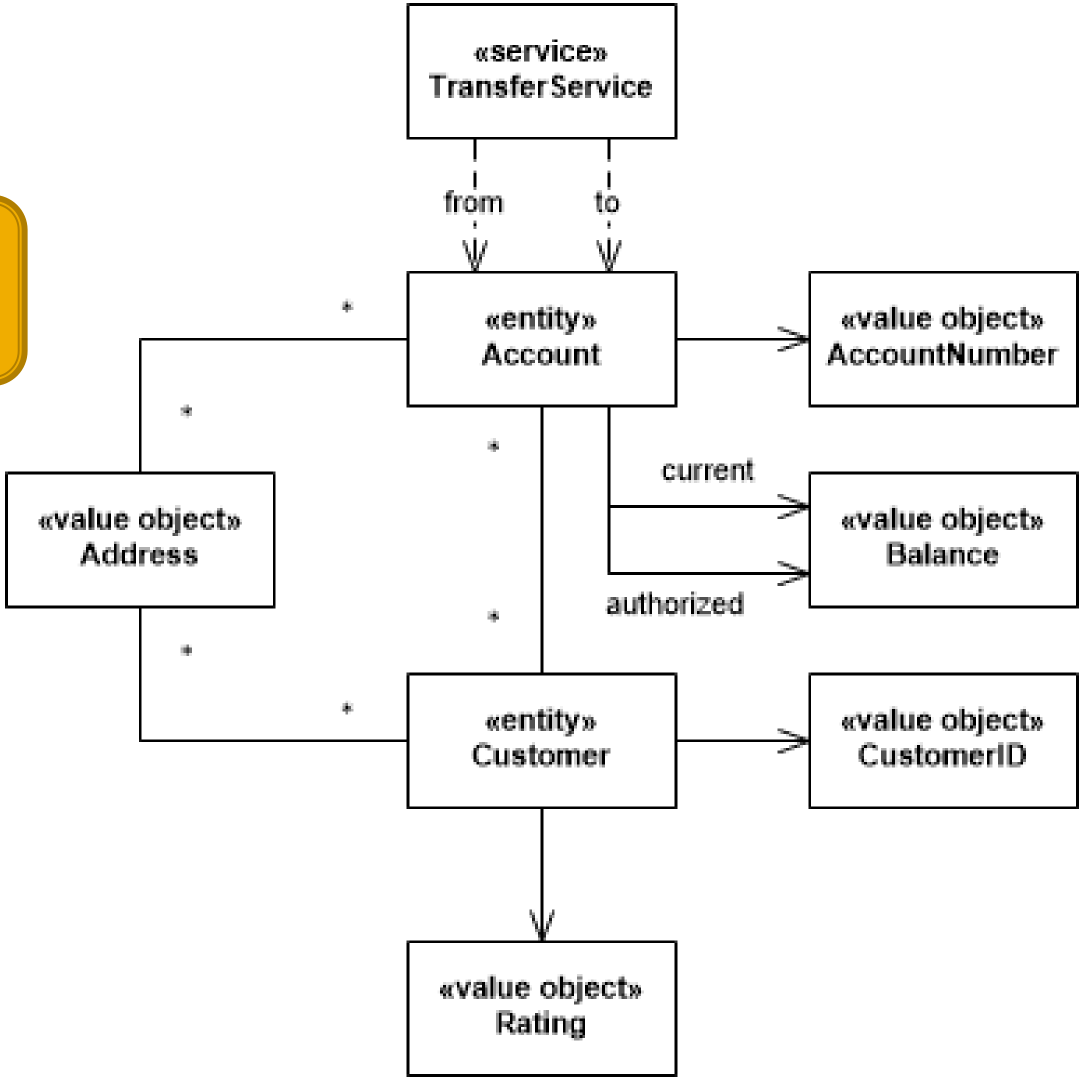
EAPLI

# DDD: agregados

Paulo Gandra de Sousa
pag@isep.ipp.pt

# An Example domain

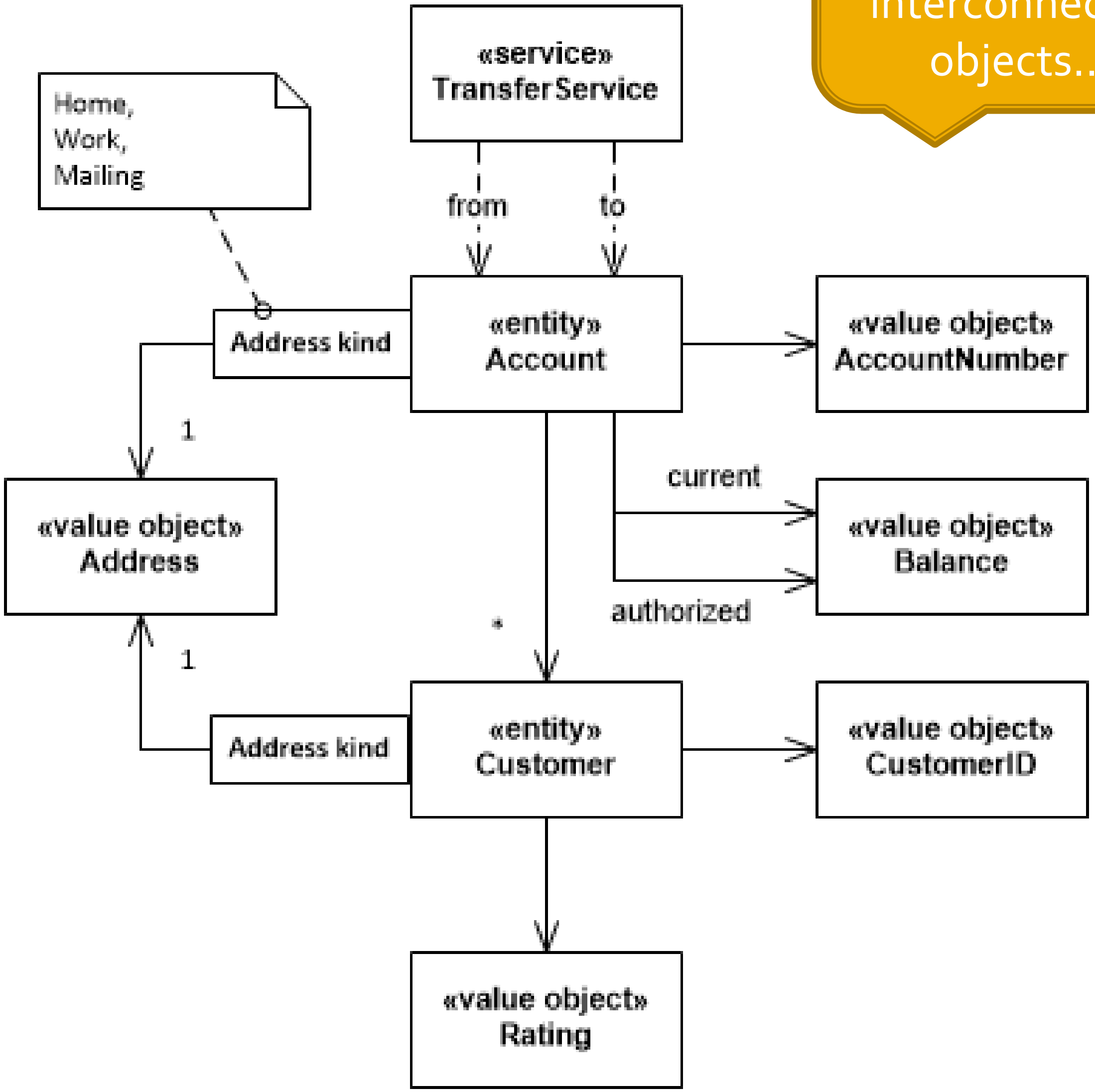


N-to-M relationships are hard

# A pragmatic design

- Remove unnecessary associations
- Force traversal direction of bidirectional associations
- Reduce cardinality by qualification of the association

we are still left with a tangle of interconnected objects…



# Aggregate

DDD

- Some objects are closely related together and we need to control the scope of data changes so that invariants are enforced

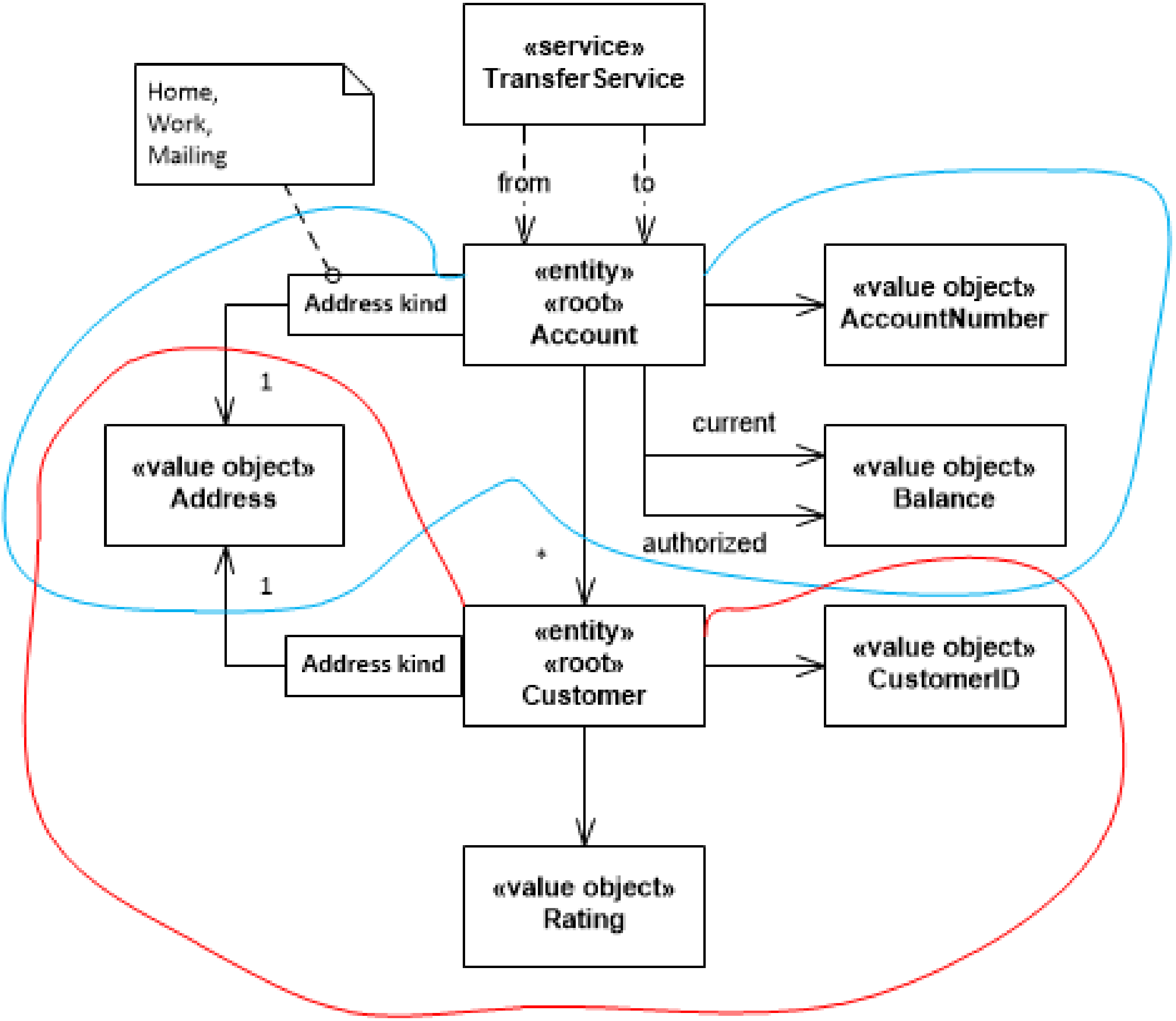
- **Therefore**
  - Keep related objects with frequent changes bundled in an aggregate
  - control access to the “inner” objects thru one single “root” object

# A more pragmatic design

Legend:

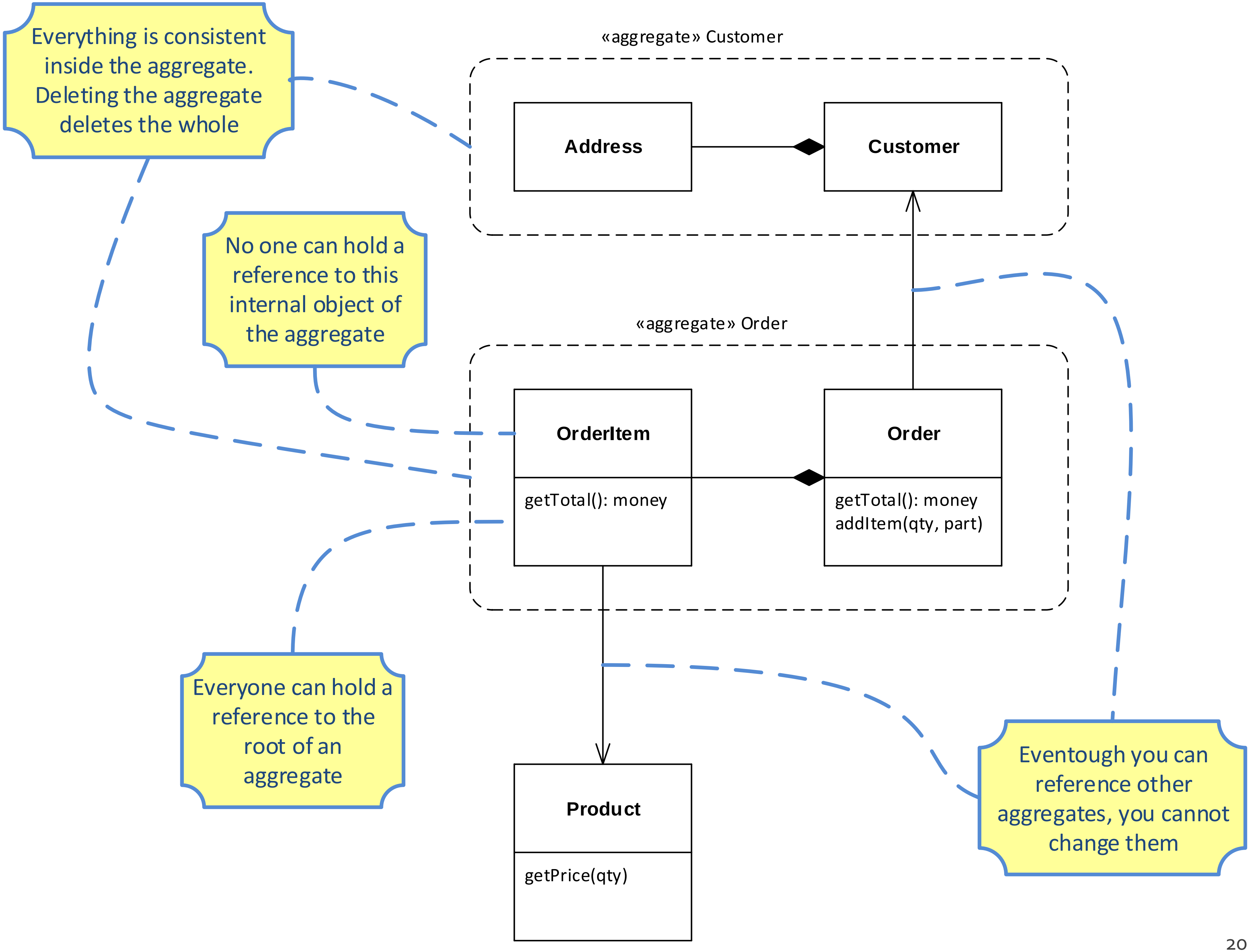
- Account aggregate
- Customer aggregate

*Address* is a value object so it can be freely shared among several aggregates



# Aggregate’s rules

- The root Entity has **global identity,** entities inside the boundary have **local identity,** unique only within the Aggregate.
- **Nothing** outside the Aggregate boundary **can hold a reference** to anything inside
- **Only Aggregate Roots** can be obtained **directly with database queries**. Everything else must be done through traversal.
- A delete operation must **remove everything** within the Aggregate boundary **all at once**.
- When a change to any object within the Aggregate boundary is committed, **all invariants** of the **whole Aggregate** must be satisfied.



Everything is consistent inside the aggregate. Deleting the aggregate deletes the whole

No one can hold a reference to this internal object of the aggregate

Everyone can hold a reference to the root of an aggregate

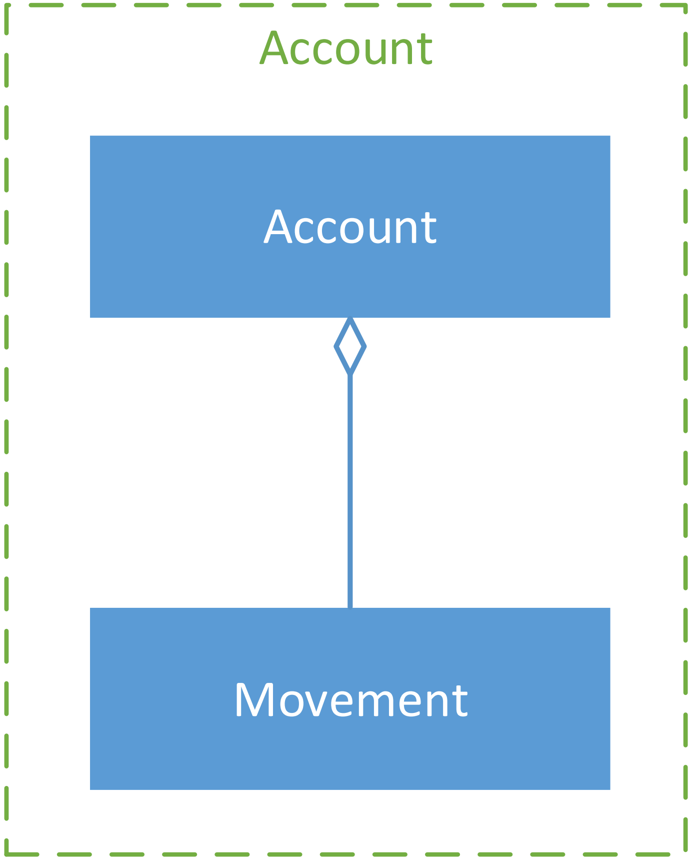
Eventough you can reference other aggregates, you cannot change them

# Aggregate boundaries

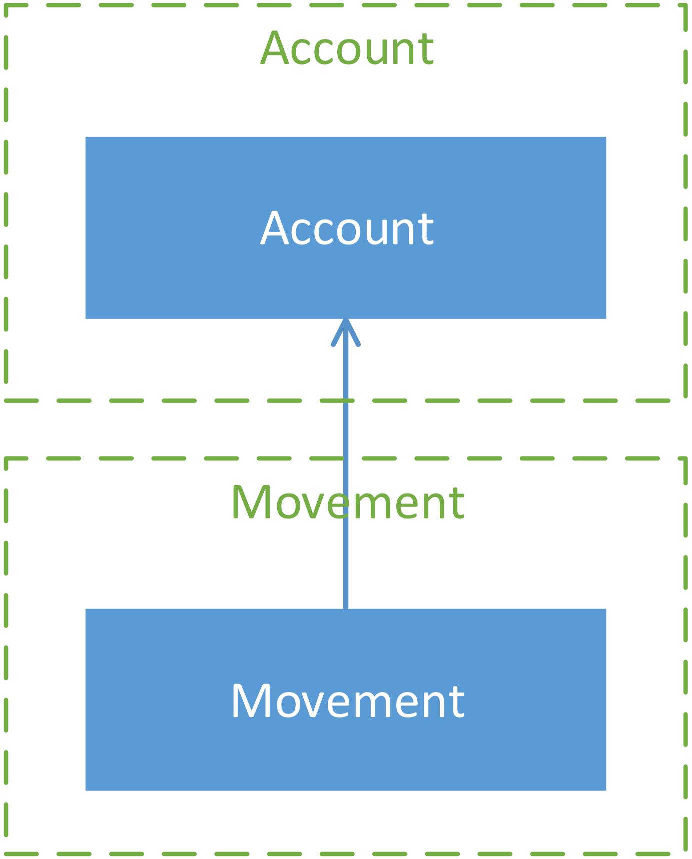
- Efficient aggregate design is hard

- Imagine the relationship between an account and its movements
- Are movements part of the Account aggregate?

# Aggregate Boundaries



vs.



# Aggregate Boundaries

- Memory consumption?
- Access concurrency?
- Data consistency?

Effective Aggregate Desi ×

Paulo

Secure | https://vaughnvernon.co/?p=838

# VAUGHN VERNON: SOFTWARE CRAFTSMAN

# Effective Aggregate Design

POSTED ON OCTOBER 9, 2014 //

This is a three-part series about using Domain-Driven Design (DDD) to implement Aggregates. Clustering Entities and Value Objects into an Aggregate with a carefully crafted consistency boundary may at first seem like quick work, but among all DDD tactical guidance, this pattern is one of the least well understood. This essay is the basis for Chapter 10 of my book, ***Implementing Domain-Driven Design.***

The documents are available for download as three PDFs and are lice **ns Attribution-NoDerivs 3.0 Unported License**.

Effective Aggregate Des ×
DDD_COMMUNITY_ESS ×
Paulo
Secure
https://vaughnvernon.co/wordpress/wp-content/up...
DDD_COMMUNITY_ESSAY_AGGREGATES_PAR... 1 / 6

**Effective Aggregate Design**
**Part I: Modeling a Single Aggregate**

Vaughn Vernon: vvernon@shiftmethod.com

Clustering **entities** and **value objects** into an **aggregate** with a carefully crafted consistency boundary may at first seem like quick work, but among all [DDD] tactical guidance, this pattern is one of the least well understood.

To start off, it might help to consider some common questions. Is an **aggregate** just a way to *cluster* a graph of closely related objects under a common parent? If so, is there some practical limit to the number of objects that should be allowed to reside in the graph? Since one **aggregate** instance can reference other **aggregate** instances, can the associations be navigated deeply, modifying various objects along the way? And what is this concept of *invariants* and a *consistency boundary* all about? It is the answer to this last question that greatly influences the answers to the others.

There are various ways to model **aggregates** incorrectly. We could fall into the trap of designing for compositional convenience and make them too large. At the other end of the spectrum we could strip all **aggregates** bare, and as a result fail to protect true invariants. As we'll see, it's imperative that we avoid both extremes and instead pay attention to the business rules.

***Designing a Scrum Management Application***

The best way to explain **aggregates** is with an example. Our fictitious company is developing an application to support Scrum-based projects, *ProjectOvation*. It follows the traditional Scrum project management model, complete with *product, product owner, team, backlog items, planned*

The concepts of this domain, along with its performance and scalability requirements, are more complex than any of them have previously faced. To address these issues, one of the DDD tactical tools that they will employ is **aggregate**.

How should the team choose the best object clusters? The **aggregate** pattern discusses composition and alludes to information hiding, which they understand how to achieve. It also discusses consistency boundaries and transactions, but they haven't been overly concerned with that. Their chosen persistence mechanism will help manage atomic commits of their data. However, that was a crucial misunderstanding of the pattern's guidance that caused them to regress. Here's what happened. The team considered the following statements in the **ubiquitous language**:

- Products have backlog items, releases, and sprints.
- New product backlog items are planned.
- New product releases are scheduled.
- New product sprints are scheduled.
- A planned backlog item may be scheduled for release.
- A scheduled backlog item may be committed to a sprint.

From these they envisioned a model, and made their first attempt at a design. Let's see how it went.

**First Attempt: Large-Cluster Aggregate**

## Original English Edition

**Effective Aggregate Design: Part 1**
**Effective Aggregate Design: Part 2**
**Effective Aggregate Design: Part 3**

## French Translation

# Transactions and consistency

**ACID**

- **A**tomic
- **C**onsistent
- **I**solated
- **D**urable

**BASE**

- **B**asic **A**vailability
- **S**oft state
- **E**ventual consistency

Inside an aggregate – ACID
Outside of an aggregate – BASE

One use case should only update **one** aggregate