ECD3 Domain Objects Design Guide

Susanne Braun
Document Revision 1.2

Licence

Document Revision: 1.2, November 2021

Acknowledgments

I would like to in particular thank Eberhard Wolff who reviewed one of the early versions of this guide and provided valuable feedback on DDD and Event Sourcing thereby contributing to notable improvements of this document. Also, I would like to thank the participants of my action research study at Fraunhofer IESE: without your feedback I wouldn't have been able to distill all the lessons learned from industry projects and case studies into concrete design patterns and best practices. Warm thanks also to all the people attending my talks and workshops, thanks for participating in my surveys, for your questions and your feedback! All of that helped me a lot to further refine the ECD3 methodology.

Table of Contents

1 Introduction

What is this all about?

This is the ECD3 Domain Objects Design Guide. Building on the Domain-Driven Design (DDD) [Eva04] philosophy it will help you design domain objects of data-intensive systems in an optimized way and assist you in properly addressing concurrency-control-related design challenges. You think this should be the job of the database and the infrastructure? Well, this used to be true for quite some time. But data-intensive systems usually call for high availability and scalability, which requires data to be distributed and replicated across different computing nodes (usually in the cloud). There is no longer one central system that could mask all the nasty details and pitfalls of concurrently accessing distributed and replicated data. As a consequence, a lot of complexity related to concurrency control and consistency is shifted from the infrastructure into the application. And you as software architect and developer will have to deal with it. Based on a taxonomy of different types of replicated data, this guide provides best practices and design patterns for an optimized design of domain objects. Keeping this classification and best practices in mind will help you design your domain objects in an optimized way so that the occurrence of conflicting updates and potential concurrency anomalies is minimized.

How to use this guide?

This guide is structured as follows: We will first recap the basic concepts, such as ACID transaction guarantees, Eventual Consistency, Domain-Driven Design and the CALM Theorem. If you're already pretty familiar with these concepts you can directly go to Section 3 to learn about the prerequisites required by the ECD3 design methodology. In Section 4 we give an extensive introduction into the ECD3 taxonomy of replicated data. Based on this taxonomy we provide best practices and design patterns for domain objects design in Section 5 and Section 6 respectively. In Section 7 you can find a cheat sheet summarizing the most important take-aways of this guide.

2 Basic Concepts

The trend towards higher distribution to exploit increased parallelism poses new concurrency and consistency related design challenges for software engineers. In this section, we provide an overview of the engineering support already available to date.

2.1 ACID Transaction Guarantees

ACID transactions [HR83] provide extensive guarantees. Proper use of transaction isolation levels [BBG+95] can ensure that concurrent data access is equivalent to serial execution (serializability), completely eliminating the possibility of concurrency and consistency anomalies.

ACID stands for:

- **Atomicity**: "A transaction's changes to the state are atomic: either all happen or none happen" [GR93].
- **Consistency**: "A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state. This requires that the transaction be a correct program." [GR93]
- **Isolation**: "Even though transactions execute concurrently, it appears to each transaction, T, that others executed either before T or after T, but not both." [GR93]
- **Durability**: "Once a transaction completes successfully (commits), its changes to the state survive failures." [GR93]

2.1.1 ACID Isolation

The term isolation is in the database research community a synonym for concurrency control. The idea is that, if every transaction would run in isolation (one transaction after the other), every transaction would only see the consistent state left behind by its predecessor [GR93]. Of course, during transaction execution data might be temporarily in an inconsistent state as updating transactions usually have to execute a number of different updates to reach a consistent state (especially if data is denormalized). Yet, it is assumed that transaction programs are correct and ensure that at the end of the transaction, the data is in a consistent state with all invariants being met (transactionally consistent state). "However, if several transactions execute concurrently, the inputs and consequent behavior of

some may be inconsistent, even though each transaction executed in isolation is correct. Concurrent transaction execution must be controlled, so that correct programs do not malfunction." [GR93] This is basically the task of isolation: "giving the illusion that each transaction runs in isolation" [GR93].

2.1.2 ACID Consistency

Due to the isolation guarantee of ACID transactions all changes of a transaction become visible simultaneously at transaction commit. Thus, atomicity is not only granted in the sense of all-or-nothing, but also in the sense that a transaction's changes are executed as one indivisible unit taking effect in one instant at transaction commit. Hence, isolation together with atomicity build the foundations of consistency in the original sense of ACID: Given that the transaction program itself is correct, data can always be in a consistent state, as related updates can be executed – from the point of view of the application – in isolation and in one instant. Notice that ACID consistency's primary focus is on the internal consistency of application data, including the validity of application invariants and constraints.

2.2 Global Transactions for Distributed Systems

Distributed transactions bring ACID guarantees to distributed systems. A distributed transaction is composed of a number of sub-transactions executed on different systems of the overall system. The commit operation of a distributed transaction is coordinated by a global transaction manager using the 2-phase-commit protocol which ensures that the distributed transaction, and all its sub-transactions commit synchronously. However, distributed transactions are barely relevant in practice today as the synchronous, blocking nature of the 2-phase-commit protocol exhibits poor performance characteristics and is also hindering qualities such as high availability. The failure of one system negatively impacts availability of all other systems participating in the distributed transaction, as these will be blocked [Ric, Hoh04].

2.2.1 Sagas

An alternative concept are sagas [GS87], permitting sub-transactions to commit independently. Thus, sagas relax the atomicity and isolation guarantees of ACID. In case the overall saga needs to be rolled back, compensating sub-transactions are executed for each already committed sub-transaction. Sagas are a special form of chained transactions originally developed for long running transactions and workflows. Developers need to implement compensating sub-transactions for each sub-transaction so

that in case of a failure the saga can be rolled back as a whole. Interestingly, in practice compensation is often much closer to the actual domain semantics than synchronous execution of distributed transactions [Hoh04]. However, sagas trade off the isolation guarantee of ACID for increased availability. As sagas do not realize consistency with serializability [WVKG08, GS87], applications might observe concurrency anomalies. Richardson presents some of the concurrency anomalies sagas might observe, and also discusses potential workarounds [Ric]. Unfortunately, he does not cover the full list of potential anomalies, and misses for example write skews [BBG+95].

Summarizing, global transactions provide an "all-or-nothing" guarantee for related updates that need to be executed in different nodes of a distributed system, in order to maintain overall ACID consistency. Distributed transactions solve this with synchronous execution of all sub-transactions (synchronous all-or-nothing), sagas with asynchronous execution of sub-transaction, and in addition – if necessary – compensating sub-transactions (eventual all-or-nothing). In case application data is (fully or partly) replicated to different nodes of the distributed system ("shared state" in software engineering parlance), the realization of any sort of global transaction becomes in particular sophisticated as we will discuss in the following.

2.3 Transactional Replication

Regarding replicated data, the database research community originally also focused on ACID transaction guarantees and serializability, referring to it as "one-copy-serializability" [BHG87]. In practice it turned out, that transactional replication has only limited horizontal scalability. In 1996, Gray et al. demonstrated with an analytical model that "a tenfold increase in replication nodes and traffic gives a thousand-fold increase in deadlocks or reconciliations" [GHOS96]. Gray eta al. concluded "that transactional replication is unstable as the workload scales up" [GHOS96]. This limitation is valid until today, except for only a few databases offering ACID transactions also on replicated data, e.g., Google Spanner [CDE+13, Bre17]. However, if Spanner replicates data across different regions, remote replicas are read-only and serve potentially stale data [LLC], which means that ACID isolation guarantees are impaired.

2.4 Non-Transactional Distributed Systems

In the distributed systems research community, Eric Brewer proposed BASE, demanding to "forfeit C and I of ACID for availability, graceful degradation and performance" [Bre00]. According to Brewer "ACID and BASE represent two design philosophies at opposite ends of the consistency-availability spectrum" [Bre12]. BASE clearly favors availabil-

ity which has top priority. It is important to note that in Brewer's trade-off discussion between "consistency" and availability, the term "consistency" refers to strong consistency, and not to ACID consistency. ACID consistency is more far-reaching than strong consistency, and includes not only strong consistency, but also guarantees that domain invariants and constraints are met at any time. [Bre00]. BASE stands for:

- **Basically Availaible**: Availability has highest priority.
- **Soft State**: Stale data is tolerable for short periods.
- **Eventual Consistency**: All replicas will eventually converge to identical state.

2.4.1 Eventual Consistency

BASE favors eventual consistency, a special form of weak consistency providing the minimum required guarantee of any practical system, namely that of convergence. The term eventual consistency was originally coined by Terry et al. in 1994 [TDP+94]. A more recent definition formulated by Terry is given in Definition 2.1.

Definition 2.1: Eventual Consistency

"A system providing eventual consistency guarantees that replicas would eventually converge to a mutually consistent state, i.e., to identical contents, if update activity ceased. Naturally, ongoing updates may prevent replicas from ever reaching identical states, especially in a mobile system where communication delays between replicas can be large due to intermittent connectivity. Thus, a more pragmatic definition of eventual consistency is desired." [Ter08]

Practically, a system provides eventual consistency if (1) each update operation is eventually received by each replica, (2) non-commutative updates are performed in the same order at each replica, and (3) the outcome of applying a sequence of updates is the same at each replica. [Ter08]

Eventual consistency propagates updates only asynchronously. Unlike strong consistency, eventual consistency does not guarantee that the last update is always returned to any subsequent access [Ter08]. During inconsistency windows, applications might read stale data or even update the same data concurrently, resulting in conflicts and potential concurrency anomalies if not properly handling by the application. Implementation of custom code for conflict resolution is a huge source of human error [Bra17, Kle16].

Conflict Handling

Rahm et al. distinguish for different types of conflict handling strategies [RSS15] illustrated in Figure 2.1. In the following we will describe these four types specified by Rahm et al. [RSS15]:

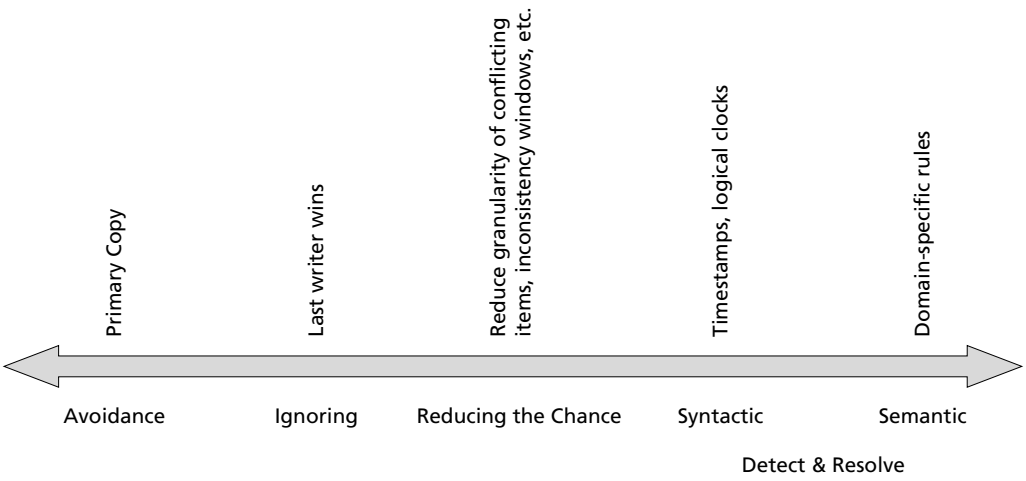

Figure 2.1: Conflict handling strategies according to Rahm et al. [RSS15]

Source: Figure from Rahm et al [RSS15] with some modifications

- **Avoidance of conflicts:** Conflicts can be completely avoided by having a dedicated replica with the exclusive permission to perform updates (primary copy). The primary copy can take care of globally serializing updating transactions. This might negatively impact availability and also results in a pessimistic approach.
- **Ignoring conflicts:** Conflicts can simply be ignored by always letting the update with the greatest timestamp win (last writer wins aka "Thomas-Write-Rule"). This strategy works only in combination with state-based update propagation. Unfortunately it is prone to lost updates.
- **Reducing the chance for conflicts:** The probability for the occurrence of conflicts can be lowered by reducing the granularity of data items upon which conflicts are tracked. Another measure is to exchange updates more frequently to reduce the duration of the inconsistency window, and thus chances for conflicting updates.
- **Syntactic conflict handling strategies:** Syntactic strategies use syntactic properties for the detection of conflicts, e.g. timestamps and logical clocks. In combination with state-based update propagation syntactic properties, such as timestamps or the replica ID, can be used to each determine a winner version of multiple conflicting versions. If

the timestamp is used, last writer wins or first writer wins are possible options. Similar to ignoring of conflicts the strategy can lead to concurrency anomalies.

– **Semantic conflict handling strategies:** Semantic strategies apply domain specific rules for detection and resolution of conflicts. For the resolution domain-specific merge functions can be implemented, as done by Amazon to resolve conflicts on the shopping card [Vog09]. Implementation of custom merge logic can be challenging and can lead to undesirable side-effects. For example, Amazon's merge function on the shopping card can lead to re-appearance of deleted shopping card items.

Note that replicas also need to establish consensus on the resolution of conflicts to achieve the convergence guarantee of eventual consistency.

To summarize, avoidance of conflicts using primary copy replication is a good option if high availability and horizontal scalability is not required.

Ignoring of conflicts and syntactic strategies are frequently used in practice in combination with state-based update propagation. The disadvantage is that these strategies are prone to concurrency anomalies, such as lost updates.

More promising seem measures that reduce the chances for conflicts from happening in the first place. This can be achieved by fine-tuning of configuration settings such as the update propagation interval, or by an optimized design of replicated data items in terms of finer granularity. This guide is specifically aimed at an optimized design of domain objects yielding in a significantly reduced chance for conflicts.

2.4.2 Causal Consistency

Causal consistency is a special subclass of eventual consistency with high practical relevance. Different definitions of causal consistency exist, e.g. in [Ter08, Vog09, ANB+95]. However, the central idea of causal consistency is to preserve causal dependencies between different update operations, and to ensure that update operations with causal dependencies are always observed in the proper order by application programs. Except from blind writes, the outcome of write operations is usually dependent on what the application has read before. The values observed by read operations in turn depend on the write operations that have been executed before. Thus, there is a potential causal relationship between a new update operation and the previous update operations that have been executed before on the replica. Douglas Terry explains it like this: The "new update is said to causally follow all of the previously received updates. A causally consistent replicated system ensures that if update u_2 follows update u_1, then a user is never allowed to observe a replica that has performed update u_2 without also performing update u_1" [Ter08]. For example, a comment to

a task description in a task management system should only be observed if also the task itself can be observed [Ter08]. If two updates are performed concurrently, that is, without knowledge of each other, then they can be incorporated in different replicas in diverging observable orders [Ter08].

Happens-Before-Relationship

A potential causal relationship between distinct write operations is dubbed the "happens-before-relationship". Practically the question of whether a particular write operation happened before another one can be decided based on the use of version vectors [JPR+83]. Version vectors are a special form of vector clocks [Mat88, Fid87] which are in turn based on Lamport Timestamps [Lam78]. It is important to note that the happens-before-relationship is only a partial ordering that does not say anything about the execution order or the reconciliation of concurrent update operations. Accordingly, causal consistency does not give any guarantees related to concurrent update operations. The main challenge of reconciliation of concurrent and conflicting update operation remains unsolved: How can in retrospect, concurrent update operations that causally depend on stale application state, be merged in such a way that for each of these concurrent update operations, the original intentions of the users are preserved?

2.4.3 Weak Consistency

Both eventual consistency and causal consistency are special variants of weak consistency with high practical relevance. The survey of Viotti and Vukolić [VV16] provides an excellent overview on the numerous other forms of weak consistency.

2.4.4 Update Propagation Format

A crucial design decision of any date replication solution is that of the update propagation format [Ter08] used to propagate updates between replicas: do replicas exchange the updated data items ("state-sending" or "state-based") or the actual update operations ("operation-sending" or "operation-based")? Database management systems usually implement state-sending protocols. However, ECD3 is taking advantage of operation-sending solutions which is an enabler for exploitation of domain semantics. The ability of a data replication solution to consider domain semantics can be used for the design of domain operations that can run concurrently and conflict-free on different nodes of a distributed system. Another advantage of operation-sending protocols is that these usually generate significantly less amounts of network traffic than state-sending protocols [Ter08].

2.5 CALM Theorem

Ten years after Brewer coined the CAP theorem [Bre00], Joseph Hellerstein formulates the CALM theorem during a keynote presentation at the ACM PODS conference [Hel10a, Hel10b]). While the CAP theorem states what in general is not possible, namely having (strong) consistency and availability in the presence of network partitions, the CALM theorem is a positive result, describing what is nevertheless possible for specific problems [HA20]. The CALM theorem defines a class of problems that can be safely implemented in a distributed system, and do not require global coordination, thus can proceed to serve requests in the presence of network partitions [HA20]. Safe problems are, according to the CALM theorem, "monotonic", i.e., they compute an ever-growing set of facts ("immutable data items") and never retreat these facts. For example, accumulating set members is monotonic whereas set deletion is not monotonic. CALM stands for: Consistency As Logical Monotonicity. The definition of the CALM theorem is given in Definition 2.2 and the formal definition of a monotonic problem is given in Definition 2.3 [HA20].

Definition 2.2: CALM Theorem

A problem has a consistent, coordination-free distributed implementation if and only if it is monotonic.

Definition 2.3: Monotonic Problem

A problem P is monotonic if for any input sets S, T where $S \subseteq T$, $P(S) \subseteq P(T)$.

Alvaro notes that a large share of predicates expressible in the SQL query language are monotonic [ACHM11]. This observation is relevant for practitioners as it means that transaction programs evaluating solely monotonic predicates can safely proceed, i.e. commit, despite of being executed on snapshots with stale data. In [HA20] Hellerstein and Alvaro illustrate a monotonic predicate with the example of distributed deadlock detection. Once a transaction dependency graph has a cycle, the cycle will not disappear, as the transactions belonging to the cycle will block each other from committing. New transaction will only add new dependencies, but not resolve the cycle. Thus, the evaluation of whether there is a cycle in a distributed transaction dependency graph is monotonic. On the contrary, answering the question whether or not an object can be garbage collected based on a shared object reference graph is not safe without global coordination [HA20]. The predicate of whether a node in the object reference graph does no longer have any incoming edges is not monotonic, as new references can be added at any time at different nodes of the distributed system.

2.6 Domain Driven Design

The ECD3 domain objects design guidelines are an advancement of Domain-Driven Design [Eva14, Eva04] and provide additional patterns for the tactical design part. The guidelines aim at the development of domain models that require less global coordination and that can therefore safely be used with eventual consistency.

Domain-Driven Design is all about the achievement of one primary goal in software engineering: reducing complexity in software. DDD in particular addresses the complexity that is already inherent in the domain or the business of the user itself [Eva04]. DDD suggests that complex domain designs be based on a model – the domain model [Eva04]. Large software projects involve different stakeholder groups, and these different groups speak different business languages and formulate different requirements for the domain model. Trying to maintain one global (canonical) model often fails in practice as these models become too complex and ambiguous. Ambiguity often arises when different stakeholder groups have the same terms in their business language, but use it with different meanings. Therefore, one central concept of DDD are bounded contexts. A bounded context defines a clear boundary within which one specific unified domain model is valid. It further defines which business language is valid within this context – the so-called ubiquitous language, used in models and code and spoken by stakeholders and developers. It is important that this clearly defined language is spoken by the domain experts as well as the software engineers, and that this language is also consistently used in the model and the code. Therefore, DDD refers to it as a ubiquitous language. A bounded context is usually maintained by one team. In a microservice architecture, one bounded context usually corresponds to one microservice. Coming up with the right bounded context cut is subject to the strategic design of DDD. The tactical design deals with the design of the domain model itself. The most important concepts of the tactical design part according to Evans [Eva04, Eva14] are:

- **Entities**: domain objects with an identity and a well-defined life cycle (e.g., a customer object).
- **Value Objects**: domain objects that can only be distinguished based on the values of their attributes (e.g., a price object). Value objects should be designed to be immutable (like the String[1] class of the Java programming language).
- **Aggregates**: clusters of coherent domain objects that commonly need to be updated together. Aggregates have a dedicated root entity responsible for maintaining consistency and domain invariants within the aggregate.

[1] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html

- **Domain Operations**: methods in the domain objects that encapsulate business logic (domain logic in DDD parlance).
- **Commands**: domain operations which lead upon execution to modifications to observable state, i.e., which perform updates.
- **Services**: In cases where domain logic is not the natural responsibility of a domain object, services can be introduced.
- **Repositories**: interfaces for querying, inserting, and deleting aggregates from the database.
- **Domain Events**: domain objects representing notable events in the domain about which domain experts or users want to be notified.

The free DDD Reference [Eva14] provides a complete overview on existing DDD concepts. For a comprehensive introduction into DDD read "the blue book" of Eric Evans [Eva04].

2.6.1 DDD Aggregates

To maintain ACID consistency of domain models, DDD has the concept of aggregates [Eva04, Eva14]. Aggregates group domain objects that usually need to updated together to ensure that the domain model is "in a consistent state", and that domain invariants are met. Aggregates should therefore only be updated in the context of ACID transactions [Eva04]. Similar to bounded contexts (see Section 2.6), aggregates draw boundaries around parts of an application's internal model. Aggregate boundaries demarcate the parts of the model that are usually processed together as one unit [Ver13].

Aggregates are composed of entities (see Section 2.6) and value objects (see Section 2.6), and have a dedicated root entity responsible for maintaining consistency and domain invariants within the aggregate, as illustrated in Figure 2.2. The root entities are supposed to encapsulate the internal organization of the other domain objects. Therefore, domain objects might not directly reference domain objects of other aggregates. Instead, aggregates cross-reference each other only via the technical ID of the root entities [Ver13].

To avoid performance issues stemming from database isolation (such as lock congestion, transaction restarts because of aggregate update conflicts, etc.), DDD further suggests modeling aggregates to be only as large as strictly necessary, and preferably updating only one aggregate in the course of one transaction [Eva04, Ver13]. Another design best practices regarding the aggregate cut of a domain model is to consider the update characteristics of domain objects. Having domain objects with different update characteristics, such as different update triggers or a different update frequency, indicates that these domain objects need not be updated together, and therefore also need not be grouped within the same aggregate boundary [Ver13].

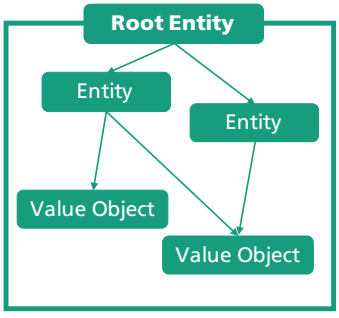

Figure 2.2: DDD Aggregate Boundary

3 Prerequisites

This guide makes some assumption regarding the programming model of your system. We briefly describe them in the following.

3.1 Programming Model

Regarding the programming model we assume that each bounded context has a three-layer architecture as illustrated in Figure 3.1. This architecture combines the originally proposed three-layered architecture of Evans with ideas of the Hexagonal Architecture introduced by Cockburn [Coc05] in order to achieve an even better separation of the business logic code from the technical code and the application code. In the application layer, application services are added to handle any application tasks as well as technical code that is not business logic, e.g., to connect third-party systems and infrastructure such as database management systems. Using the dependency inversion principle (short: DIP) [Mar03], transactions are started and ended (BOT and EOT operations in the infrastructure layer) with the start and the termination of each application service method. In [Ver13], Vernon proposes a similar three-layered architecture for implementing bounded contexts with state-of-the-practice programming languages and frameworks. For the execution of the business logic, application services delegate to the domain model hosted in the domain layer. The domain operations (queries or commands) are directly implemented in the methods of the domain objects. As DDD has the special concepts of aggregates (see Section 2.6) commands are usually implemented in the methods of root entities. In case certain domain logic is not the natural responsibility of an aggregate extra services might be introduced for provisioning of the required functionality [Eva04]. Application service methods first load the aggregates using the repositories and then invoke the method implementing the required domain operation on the domain object directly. Upon transaction commit any state changes to persistent domain objects are durably stored in the infrastructure layer.

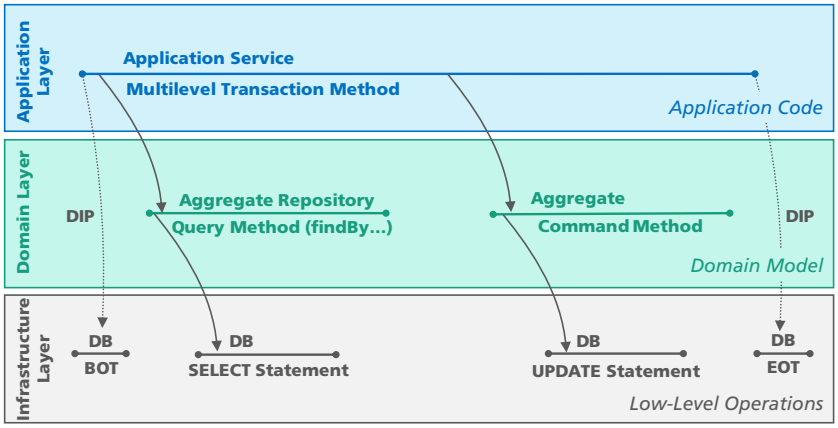

Figure 3.1: Bounded Context Three-Layer Architecture

4 ECD3 Aggregates Taxonomy

This chapter details the ECD3 aggregates taxonomy. Based on this taxonomy we provide in subsequent chapters best practices and design patterns for an optimized design of domain objects. The resulting domain models are optimized to minimize the chance for conflicts related to eventual consistency.

4.1 Definitions

Before going into the details of the taxonomy and the concrete design guidelines we first define the basic terminology. The ECD3 taxonomy is inspired by the aggregate concept of DDD [Eva14]. According to DDD aggregates are clusters of domain objects that mark boundaries of ACID consistency, and atomic execution of related updates in the context of ACID transactions (see Section 2.6.1 for an introduction to the DDD aggregate concept) [Eva14]. ECD3 relaxes this definition of aggregates and considers ECD3 aggregates. ECD3 aggregates are any self-contained clusters of persistent domain objects connected by object associations, returned as one unit by repository query operations. In order to do some meaningful business logic processing, let it be a pure query operation or an updating domain operation, most of the time a cluster of entities and value objects has to be loaded and not only a single domain object. We believe immutability is a powerful tool to solve concurrency related challenges in eventually consistent systems. The classification criteria of the ECD3 aggregate taxonomy is based on the different mutability and update characteristics of different ECD3 aggregate classes.

Definition 4.1: ECD3 Aggregate

An ECD3 aggregate is a self-contained cluster of persistent domain objects interconnected via object associations. ECD3 aggregates are loaded from the infrastructure layer into the domain layer via query operations of repositories. Updates to ECD3 aggregates are saved via repository update operations to persistent storage provided by the infrastructure layer. From the viewpoint of the domain layer ECD3 aggregates are always loaded (and persisted) as one unit from the infrastructure layer into the domain layer[a].

[a] Note that for performance reasons most ORM persistence frameworks apply lazy loading strategies, meaning that at first the cluster is not fully loaded into the domain layer, so that some objects will only be loaded upon access. However, this is transparent to the domain layer, and the domain layer can access any domain object as if the complete cluster would already have been fully loaded.

If domain models process stand-alone entities we consider this a special case of an aggregate composed of only the root entity. In the rare case domain models process single value objects, we also consider this a special single-object aggregate composed of only one value object. Figure 4.1 illustrates the different cases. Note that throughout the document root entity names are displayed in bold font.

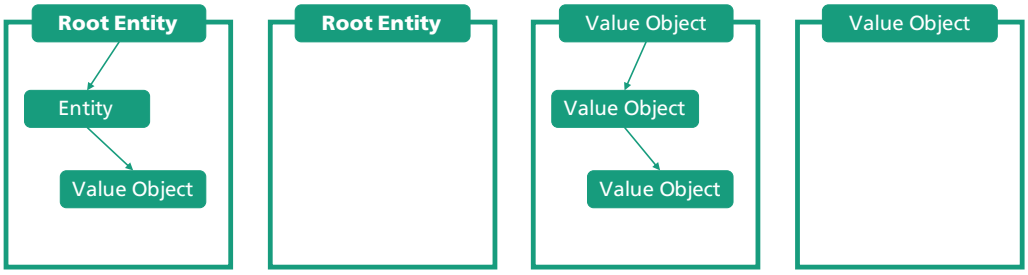

Figure 4.1: ECD3 Aggregates

Regarding the immutability discussion, we use the immutability definition of Evans, who defines it as the "property of never changing observable state after creation" [Eva04]. In contrast, an aggregate is mutable if it can have observable state changes after creation. We define observable state changes of ECD3 aggregates according to the following definition:

Definition 4.2: Observable State Change

The observable state of an ECD3 aggregate changes if the observable state of at least one of the aggregate's entities changes. The observable state of an entity changes if at least one of the entity's attributes is changed by an attribute update.

Domain-Driven Design considers the following attribute types of entities:

- primitive values (values modeled as values of primitive types)
- references to value objects (values modeled as value objects, such as Strings or composed values)
- references to entities
- references to other aggregates (by ID of the root entity of the referenced aggregate)

Depending on the attribute type, we define an attribute update as follows:

Definition 4.3: Attribute Update

A domain operation performs an attribute update if, as a result of the domain operation execution, an entity attribute has changed.

Depending on the attribute type, an entity attribute has changed if:

- primitive value: value has changed / is different
- references to value objects: values of at least one referenced value object have changed, or the number of references is different
- references to entities: referenced entities have a different identity, or the number of references is different
- references to aggregates: root entity IDs are different, or the number of references is different

For the sake of better readability we will in the following omit the ECD3 prefix when referring to ECD3 aggregates. All topics discussed in subsequent sections of this chapter are subject to ECD3 aggregates as specified according to definition 4.1.

4.2 Taxonomy

Now, we will introduce the ECD3 taxonomy and dedicate one section to each ECD3 aggregate class. Each section provides:

- an informal characterization of aggregates belonging to this class

– unique classification criteria
– illustrative examples
– guiding questions further facilitating aggregate classification

The ECD3 taxonomy distinguishes between trivial aggregates and non-trivial aggregates, as illustrated in Figure 6. We will start with the characterization of trivial aggregates.

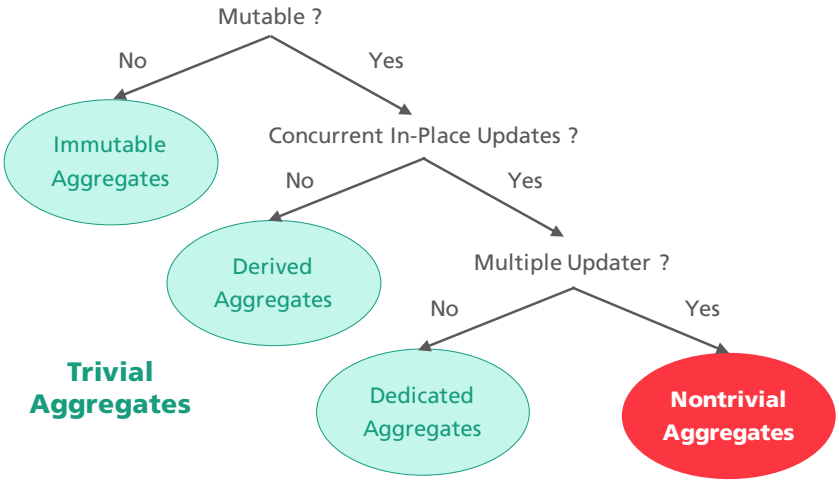

Figure 4.2: ECD3 distinguishes trivial and non-trivial aggregate classes

4.2.1 Trivial Aggregates

Trivial aggregates rule out the possibility of conflicts and are therefore trouble-free in connection with eventual consistency. Software engineers should strive to design for trivial aggregates whenever feasible and possible (see also section 5.1 for a thorough discussion of this recommendation).

The Immutable Aggregates Class

Aggregates can be designed as immutable aggregates if it can be ensured that after initialization, the aggregate will definitely not require any further changes to its observable state. An immutable aggregate can only be "changed" by being completely replaced with another aggregate of different identity. Often, immutable aggregates correspond to what Pat Helland describes as "observed facts" - facts that are observed within a system [Hel16] and never changed. These usually require being recorded for "a very long time" and are therefore not deleted; they may be archived after some time [Hel16]. A good example of observed facts are domain events that are created, recorded, and processed in the system, but are

never updated or deleted, such as an order confirmation in an online store. Instead of being updated or deleted, domain events are rather compensated by compensating domain events, like an order cancellation in the example of the online store.

According to our observations, more often than expected are business data inherently immutable and should therefore also be handled differently within the application. Once created and persisted immutable domain objects are never updated. Thus, considering the DDD aggregate concept there is usually no need to model immutable domain objects together with mutable domain objects in one aggregate cluster. However, in practice this is often done. For example, if immutable domain objects need to be frequently queried together with other mutable domain objects (because of high cohesion) those are often modelled together as one aggregate. For instance, the immutable bookings of an account are modelled as part of an overarching mutable account aggregate. In this case maintenance of a separate "Query Model" as proposed by CQRS [Fow11] can be beneficial or, even better, the usage of derived aggregates, as discussed in the next section. An in-depth discussion of the advantages and disadvantages regarding the segregation of immutable aggregates and mutable aggregates is given in section 6.2.

Classification Criteria

Immutability (no observable state changes required after initialization).

Examples

Domain Events[1]

- Order confirmations in e-commerce systems
- Incoming or outgoing goods receipts in warehouse management systems

Immutable Business Data

- A submitted offer
- The design of a survey that has gone live

Time-Series Data

- Machine sensor values in Industry 4.0 systems or predictive maintenance systems
- Stock prices in finance systems

[1]Note that Domain Events in the strict sense of DDD are a concept in its own which need to be distinguished from DDD Aggregates. In the pure teachings of DDD Domain Events are domain objects representing notable events in the domain about which domain experts or users want to be notified. ECD3 generalizes these concepts by commonly considering clusters of domain objects, and refers to them as ECD3 Aggregates. ECD3 therefore treats DDD Domain Events as one special representative of immutable aggregates.

Raw Data / Primary Data

- Tracking data used as input for recommendation systems
- Audio recordings used as input data for the training of digital assistants

Guiding Questions

If any of this questions is affirmative, this is an indicator that the aggregate is an immutable aggregate:

- Is there a domain invariant demanding immutability of the aggregate?
- Does it not make sense from a business perspective to change the aggregate once it has been initialized and persisted?
- Does the aggregate hold data that needs to be captured and recorded immutably in persistent storage for "a very long time"?

Antipattern: Pointless Use of Append-Only-Storage Principles

Note that changes to the observable state of an aggregate do not necessarily need to be implemented in a straightforward way with in-place updates. These could also be realized by means similar to "multi-versioning" applied by append-only-storage systems in order to achieve immutability of data records. Instead of loading the aggregate into the domain layer, executing attribute updates directly on the loaded aggregate (in-place update), and saving the aggregate back to persistent storage, the following can be done: A new version of the aggregate is created and initialized with the current state of the aggregate; the attribute updates are executed on the new version; the new version is persisted and returned to any subsequent read access. Applying this technique would actually result in immutable aggregate versions but not immutable aggregates. The aggregate would still have observable state changes and would thus be mutable. From the standpoint of replication, this is merely a special implementation variant for actually executing in-place updates on an aggregate. It does not help to prevent concurrency anomalies, as different versions of the same aggregate created concurrently on different replicas are still in conflict. In particular, this technique cannot prevent concurrent aggregate versions from shadowing each other's changes or even stopping the occurrence of lost updates.

How to avoid this? True immutability of aggregates indicated from the business semantics should not be confused with technical concepts developed for optimizing write access times for special types of data like time-series data in append-only stores.

The Derived Aggregates Class

Derived aggregates are mutable as state changes can be observed by the application and the user. But derived aggregates are never updated directly by concurrent in-place updates in the domain layer. Derived ag-

gregates can always be calculated on demand based on a pre-defined, unique, and side-effect-free function that does not involve any additional communication, such as establishing a global processing order at runtime, or any other dynamic consensus among replicas. The function can take as input any other required domain data of the domain model. For performance reasons derived aggregates might be pre-calculated periodically. This class is also inspired by Helland's append-only-computing pattern and corresponds to the "derived facts" in his parlance [Hel16]. Correspondingly, derived aggregates often represent "derived facts" valid at the moment or for a certain period of time, such as a weekly KPI or the current product recommendations for a user in an e-commerce shop. Another common type of derived aggregates are activities with monotonic state changes according to the calm theorem [HA20]. Examples of aggregates with monotonic state changes are the the highest bid of an online auction, or the current state of a progressive, irreversible business workflow.

Classification Criteria

Mutability (observable state changes after initialization), but no concurrent in-place updates (domain operations do not perform attribute updates directly on the aggregate). Rather, the current aggregate state is always calculated (on-demand or periodically) based on other domain data.

Examples

Calculations and Aggregations

- Data warehouse reports in data warehousing systems
- Metrics and KPIs in dashboards

Derived Views

- Alternative views of an aggregate derived from the original aggregate to serve specific request, such as requests of mobile clients
- Specific views of aggregates adjusted to meet security and data privacy policies (e.g. parts of the data has been anonymized or removed due to access restrictions)

Derived Monotonic State

If the derived aggregate represents the state of an activity, and this derived state is monotonic as defined by the calm theorem [HA20], it is an example of a derived monotonic state aggregate. If an activity has monotonic state changes this can be exploited to refactor nontrivial activity aggregates into trivial derived aggregates. We discuss this in detail in the "Derived Monotonic State Pattern" in 6.1.

- The highest bid of an online auction is monotonic as the highest bid increases monotonically with the number of incoming bids.

– The state of progressive workflows having a sequence sequence of consecutive states increases monotonically with the number of state changes.
– The number of in-stock items in an e-commerce system derived by summarizing entries from goods receipts and order confirmations as illustrated in figure 4.3[2]. The stock level is monotonic as it is based on counters for incoming and outgoing goods, and both of these counters are monotonic.

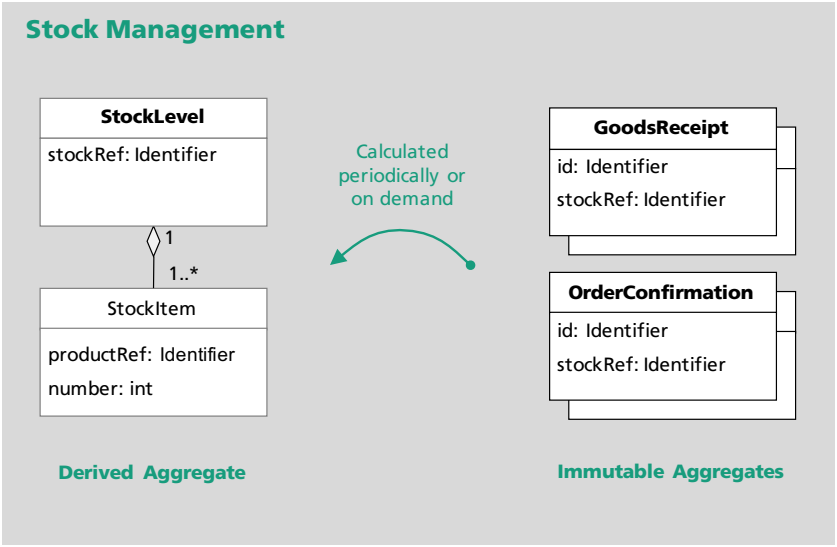

Figure 4.3: Derived aggregates in the e-commerce domain

Machine Generated Data

– Recommendations in online-shops, streaming service providers, or booking systems
– Machine Learning models in AI systems
– Predictions in forecasting systems like demand predictions, cost predictions, weather predictions, . . .

Personalized / Contextual Real-Time Data

– Timelines in social media systems
– Newsfeeds or activity streams in collaboration systems

[2]Note that ECD3 promotes a different programming style, and different aggregate modelling rules than event sourcing (assuming event sourcing as described by Fowler [Fow05b]). A thorough discussion on this can be found in section 6.1.

Guiding Questions

If any of this questions is affirmative, this is an indicator that the aggregate is a derived aggregate:

- Can the current state of the aggregate at any time be derived by evaluating other available data?
- If not, could the current state of the aggregate in principle be derived based on other data that could be collected and recorded by the system?

The Dedicated Aggregates Class

Dedicated aggregates are mutable and can receive concurrent in-place updates, but their particularity is that these concurrent updates are always issued by the same updater to which the aggregate is dedicated. An example is the settings data of a particular user account in a cloud service. The user might use different devices to access the cloud service and make changes to their settings. If one of these devices has connectivity issues or synchronizes updates infrequently, concurrent updates to the settings data might occur and can result in lost updates. This can be annoying to the user, but at least it is usually transparent to them what has happened.

Classification Criteria

Mutability (observable state changes after initialization) and concurrent in-place updates (domain operations perform attribute updates directly on the aggregate), but there is only a single updater (user or system) to which the aggregate is dedicated.

Examples

User-Generated Data

- Reviews in online shops, app stores, or travel portals
- Posts, comments, reactions in social media systems
- Chat messages in messaging applications

Dedicated Master Data

- Personal account settings in apps or cloud services
- Personal user profiles in online platforms and social media

Configuration Data

- Configuration data of a particular system in a distributed system

Guiding Questions

If any of this questions is affirmative, this is an indicator that the aggregate is a dedicated aggregate:

Table 4.1: Classification criteria of non-trivial aggregates

	Update frequency at peak times or inconsistency windows	Probability of simultaneous updates at peak times or inconsistency windows	Probability of conflicts and concurrency anomalies
Reference Aggregates	low	low	low
Activity Aggregates	medium - high	medium - high	medium - high
Collaboration Result Aggregates	medium - very high	high - very high	high - very high

- Is there only one updater (user, account, or system) that has the authority to perform updates on the aggregate?
- Do the contents of the aggregate represent "crowd data" (data generated by a large number of users that is collected and utilized by a platform to generate additional values and services)?

4.2.2 Non-Trivial Aggregates

Non-trivial aggregates are mutable and concurrently updated by multiple updaters. These characteristics are causative for the possibility of conflicts and the associated challenges in connection with eventual consistency. We classify non-trivial aggregates based on their update frequency as well as their probability of simultaneous updates, as well as their potential for conflicts and concurrency anomalies. In our experience, in practice non-trivial aggregates typically fall into the three different classes outlined in table 4.1. Next, we introduce each of these classes in more detail.

The Reference Aggregates Class

Reference aggregates are long-lived, relatively stable aggregates that are rarely updated and, if so, usually only by a small number of different updaters. Even though reference aggregates are seldom updated, they frequently contain business-critical data that is often referenced by a lot of other aggregates. Reference data is often core data of the domain and therefore also referenced by a lot of systems of the overall business domain, meaning that it is usually replicated to a lot of different systems.

Classification Criteria

The aggregate is non-trivial and therefore receives concurrent in-place updates from multiple updaters. The classification criteria of reference aggregates is given in table 4.2.

Table 4.2: Classification criteria of reference aggregates

Update frequency at peak times or inconsistency windows	Probability of simultaneous updates at peak times or inconsistency windows	Probability of conflicts and concurrency anomalies
low	low	low

Examples

Master Data

- Customer data, e.g., in customer relationship management systems
- Resources, products, and assets, e.g., in enterprise resource management systems
- Product descriptions in e-commerce systems

Values

- Valid currencies, product types, and gender

Meta Data

- Tags and descriptive data for the interpretation of raw data

Guiding Questions

If any of this questions is affirmative, this is an indicator that the aggregate is a reference aggregate:

- Does the aggregate represent classical master data that is referenced throughout the whole system?
- Is the aggregate long-lived and rarely updated?
- Is it unlikely that updates will be performed by different updaters at the same time?
- Is the aggregate a single value object or a composed value?
- Does the aggregate represent meta data?

The Activity Aggregates Class

Activity aggregates typically encapsulate the state of activities with multiple actors. Actors perform actions or sub-activities that trigger in-place updates on the state of the activity. Activity aggregates are often rather short-lived in comparison to reference aggregates, but have a lot of state changes during their lifetime. While the activity is running, this can, during peak times, result in simultaneous in-place updates on the state of the activity.

Table 4.3: Classification criteria of activity aggregates

Update frequency at peak times or inconsistency windows	Probability of simultaneous updates at peak times or inconsistency windows	Probability of conflicts and concurrency anomalies
medium - high	medium - high	medium - high

Classification Criteria

The aggregate is non-trivial and therefore receives concurrent in-place updates from multiple updaters. The classification criteria of activity aggregates is given in table 4.3.

Examples

Business Processes and Workflows

- State of an order process in an online store
- State of a package delivery process
- Booking status of limited resources like airplane seats, meeting rooms, hotel rooms
- State of an online auction

Coordination Data

- An agricultural operation performed by multiple machines and operators on the field
- State of an online Kanban board used by teams for self-organization
- Tasks in a task management system

Guiding Questions

If any of this questions is affirmative, this is an indicator that the aggregate is an activity aggregate:

- Does the aggregate represent the state of a business process or workflow that has multiple actors?
- Does the aggregate represent the availability status of a limited resource with different actors racing for that resource?
- Does the aggregate contain data that multiple actors use to coordinate some joint activity (often in real time)?
- Does the aggregate contain business operations that are not trivial (a trivial update operation would be a setter method)?
- Are there invariants associated with the activity?

Table 4.4: Classification criteria of collaboration result aggregates

Update frequency at peak times or inconsistency windows	Probability of simultaneous updates at peak times or inconsistency windows	Probability of conflicts and concurrency anomalies
medium - very high	high - very high	high - very high

The Collaboration Result Aggregates Class

Collaboration result aggregates encapsulate data produced through the collaboration of multiple updaters and usually contain the results of knowledge-generating work. Please note the interesting case that updaters can also be bots that generate, e.g., the first draft of a meeting protocol. Even though collaboration result aggregates can be either long-lived or short-lived, there are normally peak times with a very high number of simultaneous in-place updates executed on the aggregate; for example, at the end of an agile sprint when the team finishes some documentation artifact together.

Classification Criteria

The aggregate is non-trivial and therefore receives concurrent in-place updates from multiple updaters. The classification criteria of collaboration result aggregates is given in table 4.4.

Examples

Work Results

- A CAD model in a CAD system
- A software component model in a software architecture modeling system
- A machine learning model in a data science notebook
- A crop rotation plan in a smart farming system
- A spreadsheet calculation
- A whiteboard diagram

Knowledge

- Manuals and tutorials in wiki systems
- Scientific papers in collaborative text editing systems
- User story descriptions in backlog systems

Guiding Questions

If any of this questions is affirmative, this is an indicator that the aggregate is a collaboration result aggregate:

Table 4.5: Taxonomy Cheat Sheet

Aggregate	Mutable	Concurrent In-Place Updates	Multiple Updater	Update Frequency	Probability for Simultaneous Updates	Probability for Conflicts
Immutable Aggregates	–	–	–	–	–	–
Derived Aggregates	X	–	–	–	–	–
Dedicated Aggregates	X	X	–	–	low	low
Reference Aggregates	X	X	X	low	low	low
Activity Aggregates	X	X	X	medium – high	medium – high	medium – high
Collaboration Result Aggregates	X	X	X	medium – very high	high – very high	high – very high

- Does the aggregate contain work results that have been produced in a joint collaboration between different updaters (users or systems)?
- Do multiple users frequently update the aggregate at the same time?
- Is the aggregate rather large or does it have complex object associations?
- Does the aggregate contain a lot of business operations that are not trivial (a trivial update operation would be a setter method)?

4.2.3 Taxonomy Cheat Sheet

In Table 7.1 we provide an overview on the ECD3 aggregates taxonomy and its classification criteria.

5 Best Practices

Now, we provide some general modeling best practices for the design of ECD3 aggregates.

5.1 Trivial Aggregates First

Whenever technically feasible and also feasible from the viewpoint of domain semantics prefer trivial aggregates over non-trivial aggregates. Quite often activity aggregates can be refactored into derived and immutable aggregates as described in the "Derived Monotonic State Pattern" in section 6.1. But do not try to enforce this recommendation, trying to model each aggregate as either immutable, derived or dedicated. In particular, do not mix up event-sourced aggregates [Fow05b] with ECD3 derived aggregates. Each observable state change of event-sourced aggregates must be initiated by an event, which is quite restrictive, and in a lot of cases adds unnecessary complexity to your domain model. We discuss the differences of event-sourced aggregates and ECD3 derived aggregates in more detail in section 6.1.6. As a general rule of thump, use derived aggregates only if it can be clearly motivated by domain semantics. The same is true for immutable aggregates. On the other hand, if it is a natural fit exploit immutability. It is a low hanging fruit often overlooked even though there are plenty of examples such as:

- a confirmed order
- a released shift schedule
- a submitted questionnaire
- ...

More examples of immutable aggregates can be found in section 4.2.1.

5.2 Non-Trivial Aggregates? – Make Well-Informed Trade-Off Decisions

The ECD3 aggregates taxonomy provides means to reason about the chance for conflicts and consequential concurrency anomalies on certain parts of your domain model. Resulting follow-up questions such as reasoning about the consequences and costs of data corruption caused by these conflicts should be further pursued. Reasoning about those issues together with domain experts enables software engineers to make well-informed trade-off decisions between correctness and consistency of busi-

Table 5.1: Trade-off Decisions Template

Aggregate	Concurrency anomaly probability	Severity of data corruption	Costs for manually fixing corrupted data	Conflicting quality attribute	Costs for trading off conflicting quality attribute
Shopping Card	low	low	$ 250 per incident	99.999% availability	Opportunity costs of $ 40.5 Mio per year
...					
...					

ness data on the one hand, and other quality attributes of the system, such as availability and scalability on the other hand. For example, Amazon accepts certain anomalies on its shopping card service in the rare event of conflicts, thereby trading off correctness for high availability [Vog09]. Outages of the shopping card service involves considerable higher costs than sporadic manual handling of incorrectly placed orders.

The template given in table 5.1 can be used to facilitate the discussion between software engineers and domain experts. In the long run the template is also useful to establish transparency and plausibility of fundamental trade-off decisions. Note that costs provided in column four and six should be given with the reference value, such as "per year". The conflicting quality attribute traded off in order to avoid the occurrence of concurrency anomalies should be provided as a measurable value in column five.

6 Design Patterns

Now, we present detailed design patterns assisting software engineers to increase the share of trivial aggregates and minimize the chance for conflicts in eventually consistent domain models. Patterns are described in a standardized format to facilitate comparability, comprehensibility, and systematic approaching. We first describe the context in which the pattern can be applied, then describe the main forces driving the need for this pattern, next characterize the resulting solution, illustrate the solution with concrete examples in the following, afterwards we outline the advantages and disadvantages of the resulting context, and conclude with a discussion of related patterns.

6.1 The Derived Monotonic State Pattern

This pattern explains how to factor out immutable and derived aggregates out of complex non-trivial activity aggregates in order to increase the share of trivial aggregates and reduce the chance for conflicts.

6.1.1 Context

A bounded context cut driven by functionality and use cases has successfully been created. The bounded context under consideration contains non-trivial activity aggregates that are frequently and concurrently updated in-place. You expect conflicting updates on the aggregate to happen on a regular basis. In the non-replicated case this might have a negative impact on performance if the database uses a lock-based pessimistic concurrency control scheme. In case the persistence layer uses optimistic concurrency control based on versioning it might lead to transaction aborts and corresponding restarts which also might negatively impact performance. In the replicated case it can mean a lot of manual reconciliations or concurrency anomalies like lost updates if a proper reconciliation scheme is not available.

6.1.2 Forces

- Performance and availability of functionality implemented with or based on the aggregate is a crucial business factor.
- A reconciliation scheme that is "good enough" cannot be implemented with reasonable effort.

– Potentially conflicting updates cannot be implemented to be compatible or commutative.

6.1.3 Solution

Revisit the activity and in particular the associated domain events of this activity. Is it possible to factor out (1) domain events and (2) domain objects representing the current state of the aggregate? Is it further possible to calculate the current state of the activity out of domain events and other domain objects? If this is the case: implement the functionality for the calculation of the activity's state in a side-effect free function returning a derived aggregate representing the activity's current state. Implement this function to be idempotent, monotonic (see Definition 2.3) and deterministic. Place the function in the root entity of the activity aggregate. Figure 7.1 provides an illustration of the refactoring and shows how to transform complex non-trivial activity aggregates into a number of less complex aggregates: the less complex activity aggregate containing the side-effect free function, the derived aggregate representing the activity's current state, as well as one immutable aggregate for each domain event of the activity.

6.1.4 Examples

Stock Management

The stock management bounded context model of an e-commerce application could be modelled with a non-trivial stock aggregate as illustrated in figure 6.2. The Derived Monotonic State Pattern recommends reviewing this activity aggregate to check whether it is feasible to factor out corresponding domain events, as well as domain objects representing the current state of the activity. In this example goods receipts events and order confirmation events can be modelled as standalone immutable aggregates. By introducing a derived aggregate "StockLevel" the current state of the stock, in terms of availability of products, can be represented in the model (see Figure 6.3). Note that the Stock aggregate then becomes a reference aggregate encapsulating only general information about the stock like its location or its capacity. This general information is expected to be updated rarely. The refactored Stock aggregate contains a side-effect free function for calculation of the current stock level as illustrated in listing 6.1. It calculates the current stock level by considering the products sold by the e-commerce application in general, as well as corresponding goods receipts and order confirmations. An optimization could be to consider the products, a particular snapshot of the stock level aggregate and only goods receipts and order confirmations observed after creation of the stock level snapshot. Note that the derived number of available items per product is calculated based on two monotonic counter values (total number of incoming items and total number of outgoing items). The relevant

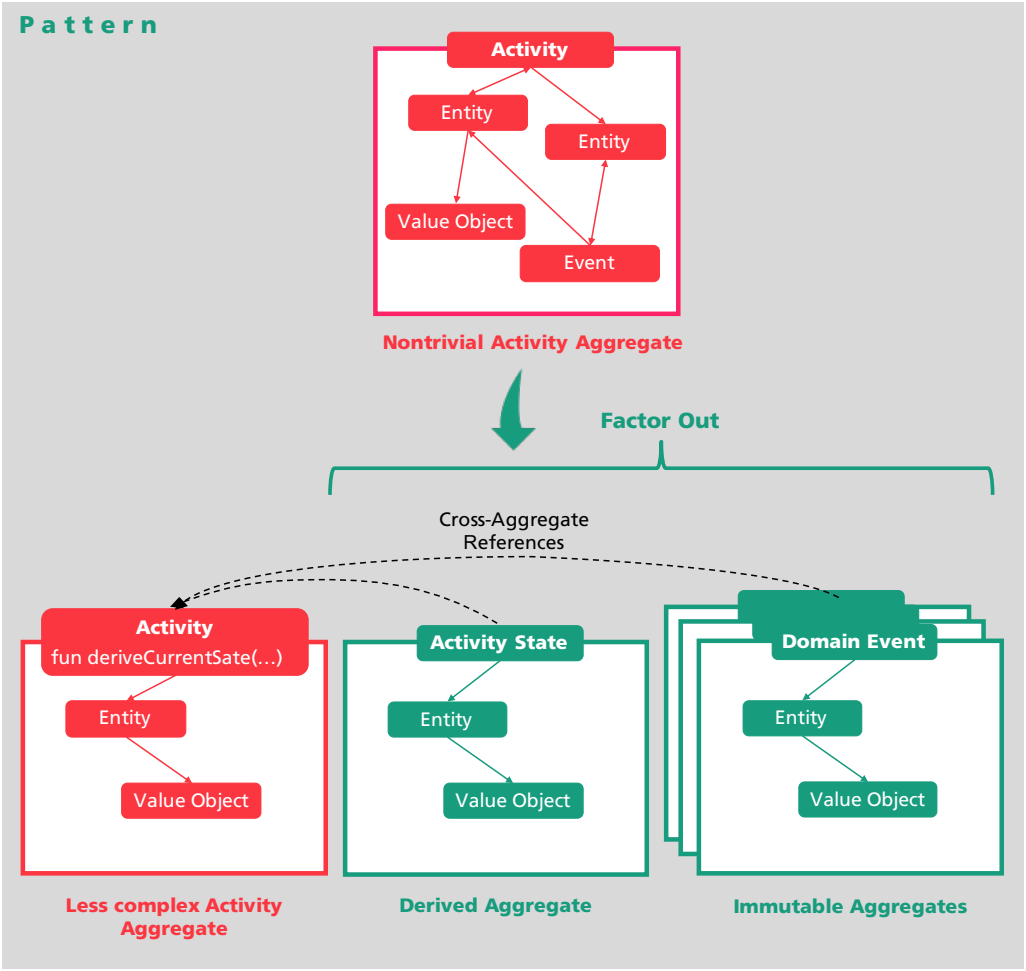

Figure 6.1: Factoring out trivial aggregates from activity aggregates

code is located in lines 14-19 and line 36 of listing 6.1). As goods receipts and order confirmations are immutable and therefore need not be updated synchronously there is no need to model "GoodsReceipts" and "OrderConfirmations" aggregates for encapsulation of all goods receipts and all order confirmations. It is sufficient to have corresponding repository interfaces (not shown in the figures) in your model which provide methods for querying the required collections of event aggregates, as well as methods for the creation of event aggregates. If additional functionality on goods receipts and order confirmations is required that is not the natural responsibility of the Stock or the StockLevel aggregates additional services can be introduced.

```
1
2  @AggregateRoot
3  public class Stock {
4
5    private Identifier id;
6
7    public StockLevel calculateStockLevel(Identifier stockId,
8        Identifier stockLevelId, Set<GoodsReceipt> goodsReceipts,
9        Set<OrderConfirmation> orderConfirmations,
10       Set<Product> productCatalogue) {
11
12     HLCTimestamp calculationTime = HybridLogicalClock.instance().getTime();
13
14     // monotonic function calculates incoming items per product
15     Map<Identifier, Integer> incomingItemsCount = countItems(
16         goodsReceipts);
17     // monotonic function calculates outgoing items per product
18     Map<Identifier, Integer> outgoingItemsCount = countItems(
19         orderConfirmations);
20
21     Set<StockItem> items = new HashSet<>();
22     for (Product product : productCatalogue) {
23       Identifier productRefId = product.getId();
24
25       int incomingCount = 0;
26       if (incomingItemsCount.get(productRefId) != null) {
27         incomingCount = incomingItemsCount.get(productRefId);
28       }
29
30       int outgoingCount = 0;
31       if (outgoingItemsCount.get(productRefId) != null) {
32         outgoingCount = outgoingItemsCount.get(productRefId);
33       }
34
35       // total available item count derived from two monotonic function values
36       int availableCount = incomingCount - outgoingCount;
37
38       items.add(new StockItem(productRefId, availableCount));
39     }
40
41     // create final return value
42     return new StockLevel(stockLevelId, stockId, items, calculationTime);
43   }
```

Listing 6.1: Derivation of the current stock level with monotonic functions

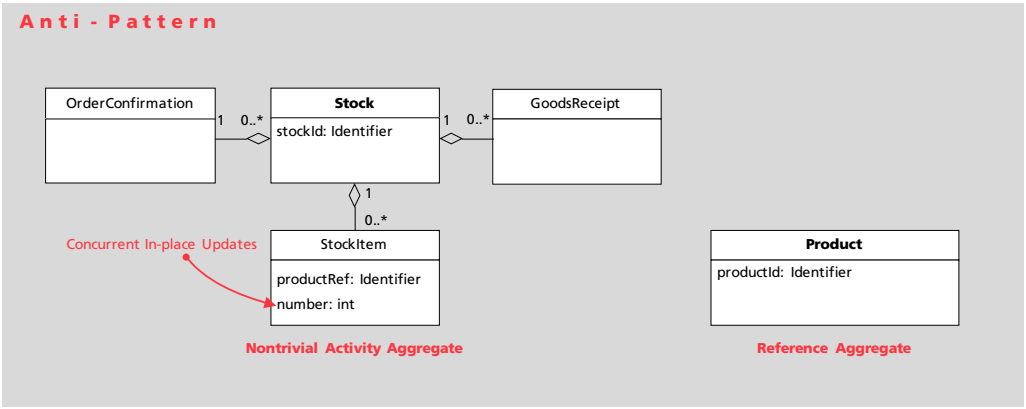

Figure 6.2: Example of the Derived Monotonic State Anti-Pattern

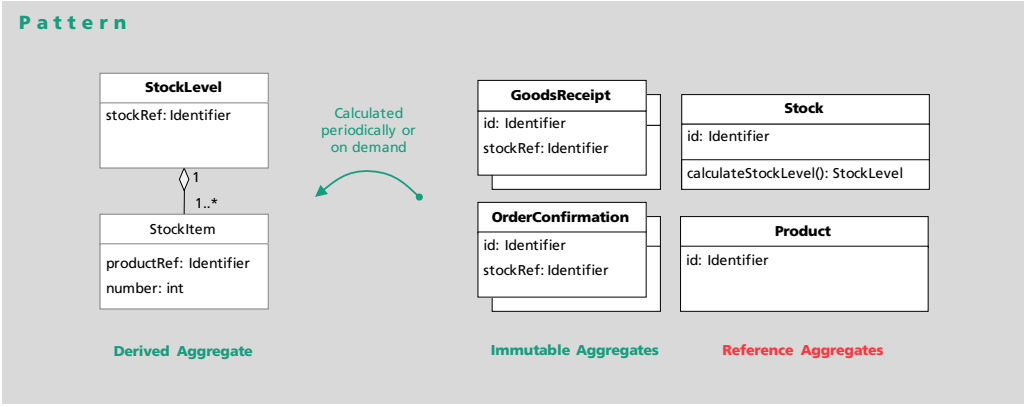

Figure 6.3: Stock Management as Example for the Derived Monotonic State Pattern

6.1.5 Resulting Context

The pattern has the following benefits:

- By factoring out derived aggregates and immutable aggregates, the share of trivial aggregates is significantly increased.
- The size of remaining non-trivial aggregates is smaller which reduces the risk for conflicts.
- The chance for conflicts on the remaining non-trivial aggregates is considerably lower if, as in the examples, activity aggregates can be refactored into reference aggregates.
- The complexity of the resulting model is in so far reduced as aggregates are consisting of fewer domain objects with fewer object associations.

The pattern has the following drawbacks:

- Architects and developers familiar with Event Sourcing [11] might consider it unnatural to model domain events as standalone aggregates or standalone domain objects.
- Modelling the activity's state as separate aggregate that is cross-referencing the original activity aggregate via ID only might seem unnatural to software architects and developers at first sight.

The following issues remain to be addressed:

- Ad-hoc calculation of derived aggregates can result in longer response times and reduced performance.
- If derived aggregates are only calculated periodically domain operations might process an outdated activity state.
- Tradeoff decision between strong consistency (ad-hoc calculation of the derived aggregate upon each query) and performance (periodic asynchronous calculation of the derived aggregate) in case aggregate derivation is performing poorly [Fow05b].

6.1.6 Related Patterns

- **CQRS** [Fow11] divides the domain model into two segregated parts: the query model and the command model. The basic idea is to have separate models for queries (domain operations that do not change the observable state of the model) and commands (domain operations that do change observable state, but do not return a value) [Fow05a]. State change requests need to be directed to the command model which processes the command and communicates the state change to the read model. There are a lot of variations, but in practice each state change must usually be initiated by issuing a command event first. Issued command events are persisted (sometimes in a dedicated command model database optimized for write access), are processed asynchronously and result in an updated query model. Issuing a command event seems to be similar to issuing a domain event at first sight. But note that both are conceptually actually two different things. Command events are issued to initiate observable state changes, whereas domain events are issued to notify about notable events domain experts or users would want to be notified of. Domain objects of the query model can be seen as a special kind of derived aggregates. However, CQRS is usually applied in general to a complete bounded context model, forcing software architects to always model updating domain operations as commands in the command model. In a lot of cases, it is quite cumbersome if domain objects cannot be updated directly in-place, but instead command objects have to be created first in order to achieve an updated aggregate state in the query model. In contrast, the advantage of the Derived Monotonic State Pattern is that it can be selectively applied to parts of

the model where it is a natural fit to derive the current state of an activity out of domain events and additional domain data (e.g. because of monotonic domain logic).

– **Event Sourcing** [Fow05b] requires that every state change of an aggregate is initiated by an event. Thus, for each aggregate update an event needs to be created first. The current aggregate state might be calculated upon each query by re-processing the complete event history of an aggregate. In addition, aggregate snapshots might be persisted in order to increase performance. Event sourcing has, similar to CQRS, an unfamiliar programming style as for each update an event has to be created and issued first. Event sourcing can be tricky and re-processing of events might have unintended side-effects in particular if event processing requires interaction with non-event sourced third-party systems [Fow05b].

  Note that events in event sourcing are also different from domain events, as event sourcing requires that for each state change an (sometimes artificial) event has to be emitted first. In comparison, the advantage of the Derived Monotonic State Pattern is the same as above: it can be selectively applied to parts of the model where it is a natural fit, e.g. to parts of the model with monotonic state changes. The advantage of monotonic state changes is that these state changes can be processed order-independently. Event sourcing on the other hand, derives aggregate state based on a private update event log that needs to be processed in chronological order. In particular, in a distributed setting it would require a global order of this event log. Therefore, if event sourced aggregates are replicated to multiple nodes of a distributed system, as proposed e.g. in [Ric], global consensus on the event log order is required. As event sourcing was originally not intended for this, it also seems to be a violation of the information hiding principle, since the event log should be private for an event-sourced object. Further note that using a message broker for establishing global order does not necessarily guarantee the absence of concurrency anomalies, as the global ordering provided by the message broker is, without further global coordination, not guaranteed to be serializable. This in turn, guaranteeing global serializability of the update event log, would be a hard problem if updates are not restricted to a single node.

  Also note that the Derived Monotonic State Pattern has slightly different modelling rules (domain events are standalone ECD3 immutable aggregates and not part of the derived aggregate).

6.2 The Segregate Aggregate Classes Pattern

6.2.1 Context

A bounded context cut driven by functionality and use cases has successfully been created. The bounded context under consideration contains

large DDD aggregates with complex object associations. Because of the size of the DDD aggregate, you expect conflicting updates on the aggregate to happen on a regular basis. In the non-replicated case this might have a negative impact on performance if the database uses a lock-based pessimistic concurrency control scheme. In case the persistence layer uses optimistic concurrency control based on versioning it might lead to transaction aborts and corresponding restarts which also might negatively impact performance. In the replicated case it can mean a lot of reconciliations or concurrency anomalies like lost updates if a proper reconciliation scheme is not available.

6.2.2 Forces

- Performance and availability of the functionality implemented with or based on the aggregate is a crucial business factor.
- A reconciliation scheme that is "good enough" cannot be implemented with reasonable effort.
- Potentially conflicting updates cannot be implemented to be commutative.

6.2.3 Solution

Revisit large DDD aggregates and check whether the domain objects of these aggregates actually belong to different classes according to the ECD3 taxonomy. If the domain objects of large DDD aggregates belong to different ECD3 classes it means that these objects have different update characteristics such as different update frequencies, different update triggers and different updaters. This in turn indicates that these domain objects most likely need not be updated synchronously and consistently. Consequently, these objects also need not be protected by the same DDD aggregate boundary. In this chase, check whether it is possible to refactor the DDD aggregate into a set of smaller ECD3 aggregates so that all domain objects of each aggregate clearly belong to the same ECD3 class. The pattern is illustrated in Figure 7.2.

6.2.4 Examples

Agricultural Jobs

Smart farming apps deliver computer-aided support for the conduction of complex agricultural jobs carried out on a number of different fields involving a lot of resources like staff members, machines, as well as complicated upfront planning of logistics. Figure 6.5 gives an excerpt of a bounded context model supporting different uses cases for planning and execution of agricultural jobs. The model contains entities representing the operation type of the job such as seeding, spraying and so on, as well

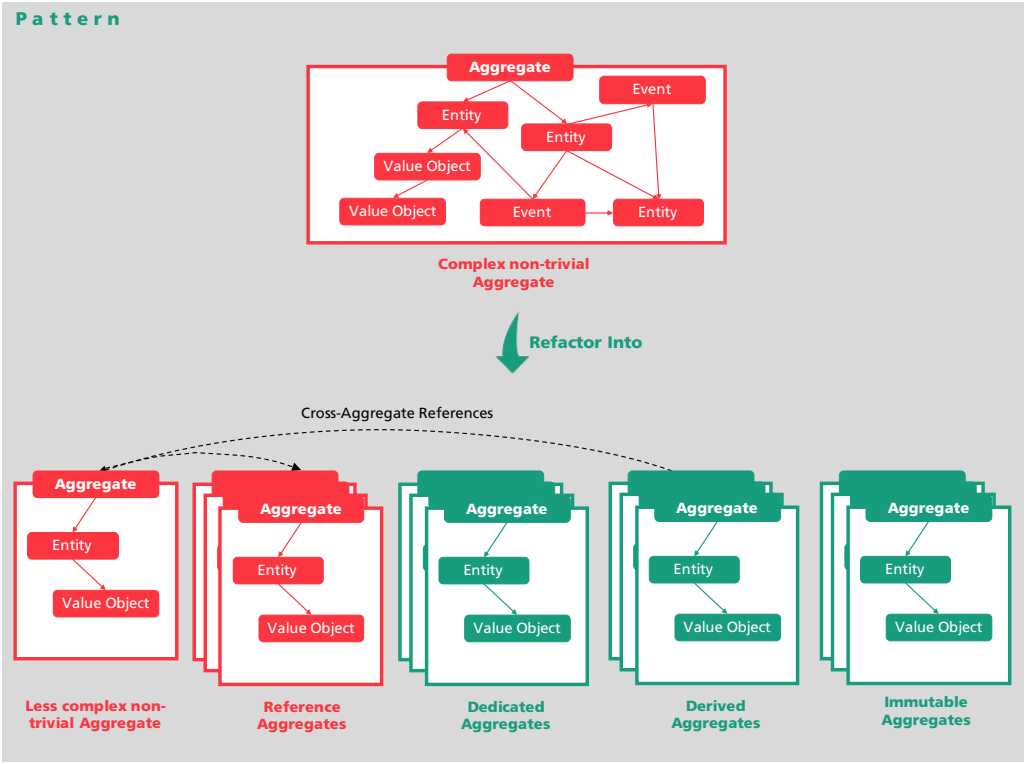

Figure 6.4: The Segregate Aggregate Classes Pattern

as affected fields, assignments of required resources and documentation records that need to be created by the operators during the conduction of the job (in Germany farmers have to document among other things the amount of fertilizers and chemicals applied to a field). Depending on the use cases the current job status, affected fields, assigned resources or documentation records are presented to the user in connection with the job. Other use cases display all conducted jobs for a particular field during the agronomic season or all jobs a machine is assigned to. This led to a design with a large aggregate having complex, (partially bi-directional) object associations that is hard to maintain and also prone to conflicting update operations.

Applying the Segregate Aggregate Classes pattern reveals that documentation records are dedicated data and therefore can be refactored into stand-alone aggregates. Fields, Machines and StaffMembers are actually reference data that is supposed to be rarely updated and should therefore be modelled as separate aggregates. The OperationType can be implemented as values (value objects) of an enumeration. The result of this refactoring is displayed in figure 6.6.

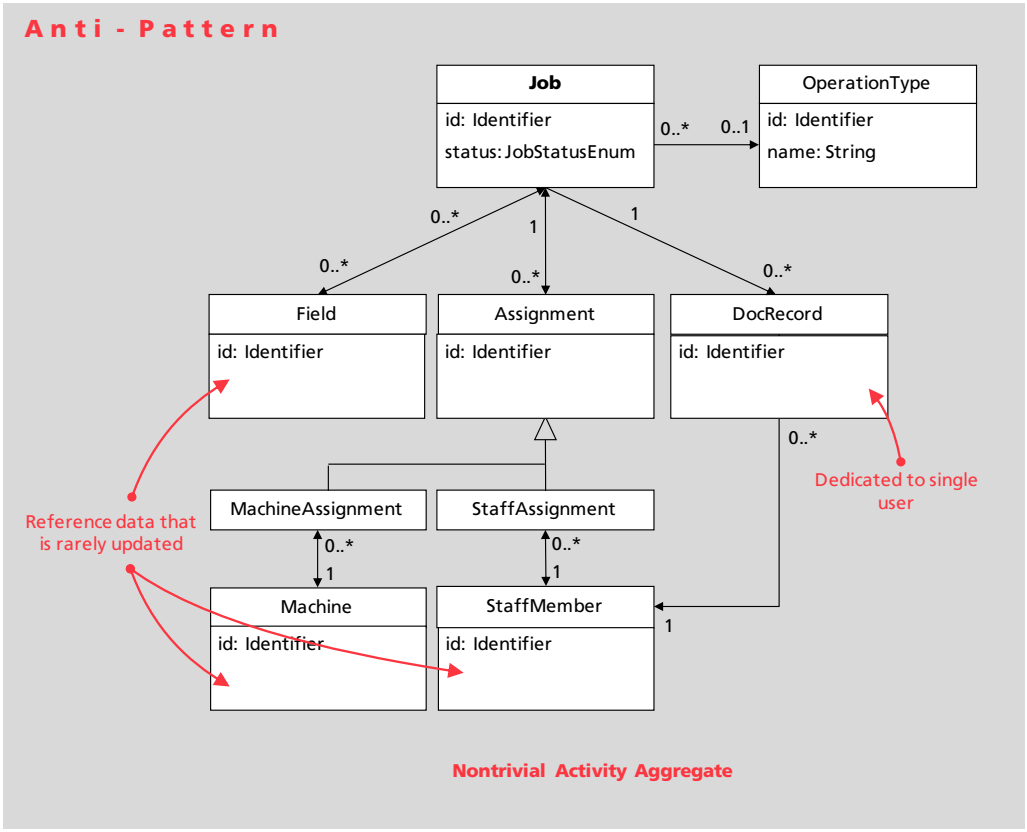

Figure 6.5: Example of the Segregate Aggregate Classes Anti-Pattern

As illustrated in figure 6.6, applying the rules of the Segregate Aggregate Classes pattern already led to an increase of trivial aggregates and a significant reduction of complexity of the bounded context model. But can we make better? During job execution we still expect conflicting updates on the job status to be issued by different operators. As operators are using mobile devices during job execution we assume high latencies and even network partitions which further increases the chances for conflicts. Applying in addition the Derived Monotonic State pattern, we can refactor the job activity aggregate into a less complex activity aggregate representing the planning status of the job, a derived aggregate representing the current execution status of the job, as well as domain events notifying about job status updates. The resulting bounded context model is given in figure 6.7 and figure 6.8.

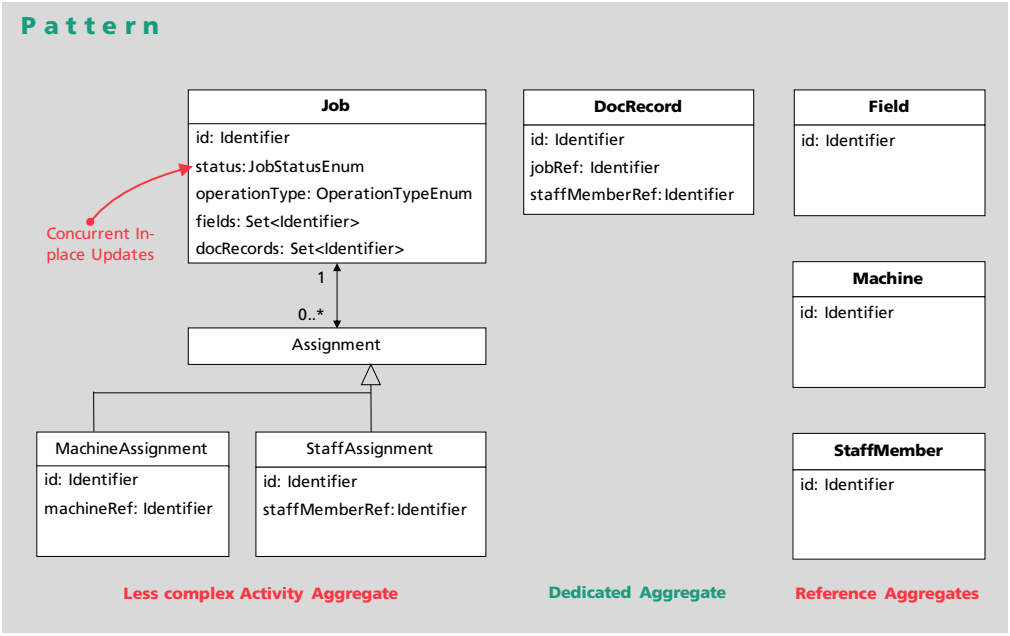

Figure 6.6: Agricultural Job Management as Example for the Segregate Aggregate Classes Pattern

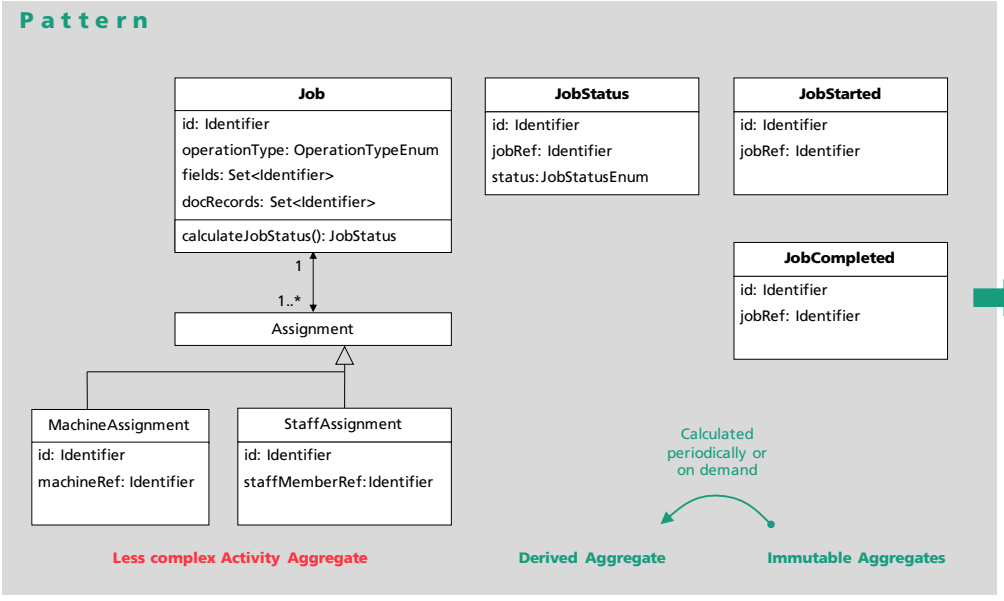

Figure 6.7: Agricultural Job Management as an Example for the Patterns: Segregate Aggregate Classes and Derived Monotonic State – Part 1

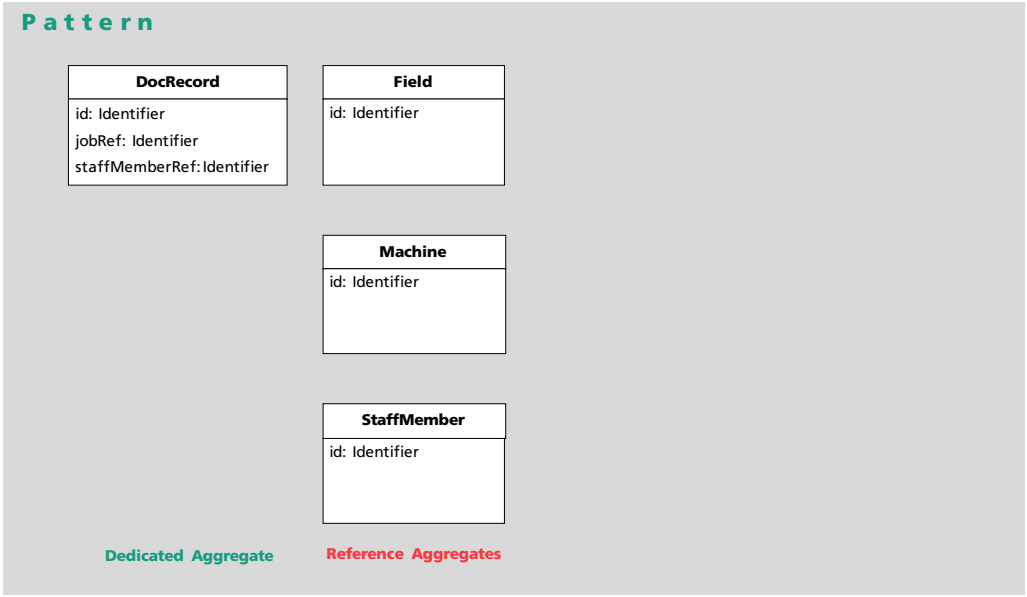

Figure 6.8: Agricultural Job Management as an Example for the Patterns: Segregate Aggregate Classes and Derived Monotonic State – Part 2

6.2.5 Resulting Context

The pattern has the following benefits:

- By segregation of domain objects belonging to different classes of the ECD3 taxonomy large aggregates can be significantly reduced in terms of size, complexity and ramification of object associations.
- As aggregates become smaller the chances for conflicting updates are reduced.
- By factoring out immutable, derived and dedicated data into standalone aggregates the advantages of trivial aggregates can be exploited as for these aggregates no reconciliation schemes are required in case of conflicting updates.
- By factoring out reference aggregates the low update frequency of reference data can be exploited. As chances for conflicting updates are low simple automatic reconciliation schemes or even manual reconciliation schemes are usually sufficient.

The pattern has the following drawbacks:

- Architects and developers used to Object Relational Mappers (ORMs) might find it inconvenient to model and code cross-aggregate references via root entity IDs instead of direct object associations.
- By segregating complex object graphs into smaller subsets developers have to implement a higher number of aggregate repositories and cor-

responding queries for loading of cross-referenced aggregates. Developers might find this less convenient and more labor-intensive.

The following issues remain to be addressed:

- Design and implementation of repositories for dynamic loading of cross-referenced aggregates.

6.2.6 Related Patterns

- The aggregate pattern of DDD [Eva04, Eva14] itself recommends drawing clear boundaries around clusters of domain objects that need to be strongly consistent and updated synchronously. Aggregate-external objects are only allowed to reference the root entity. The DDD aggregate pattern aims at avoiding models with complex object associations, as well as lower chances for conflicts and deadlocks arising with pessimistic concurrency control schemes of database management systems (note that optimistic concurrency control schemes basically convert conflicts into transaction restarts which can also have a negative impact on performance). Thus, applying the Segregate Aggregate Classes Pattern serves the original intents of DDD's aggregate pattern.

7 Cheat Sheet

Table 7.1: The ECD3 Aggregates Taxonomy

Aggregate	Mutable	Concurrent In-Place Updates	Multiple Updater	Update Frequency	Probability for Simultaneous Updates	Probability for Conflicts
Immutable Aggregates	–	–	–	–	–	–
Derived Aggregates	X	–	–	–	–	–
Dedicated Aggregates	X	X	–	–	low	low
Reference Aggregates	X	X	X	low	low	low
Activity Aggregates	X	X	X	medium – high	medium – high	medium – high
Collaboration Result Aggregates	X	X	X	medium – very high	high – very high	high – very high

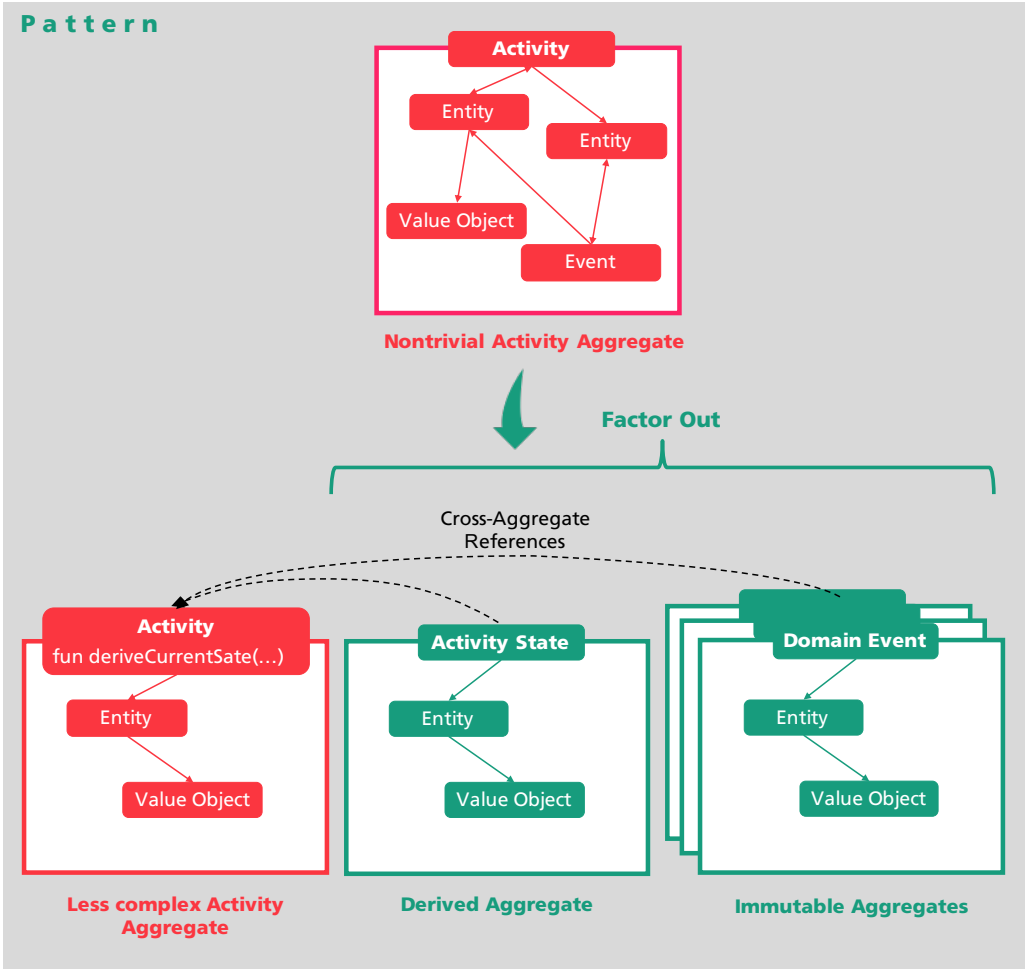

Figure 7.1: The Derived Monotonic State Pattern

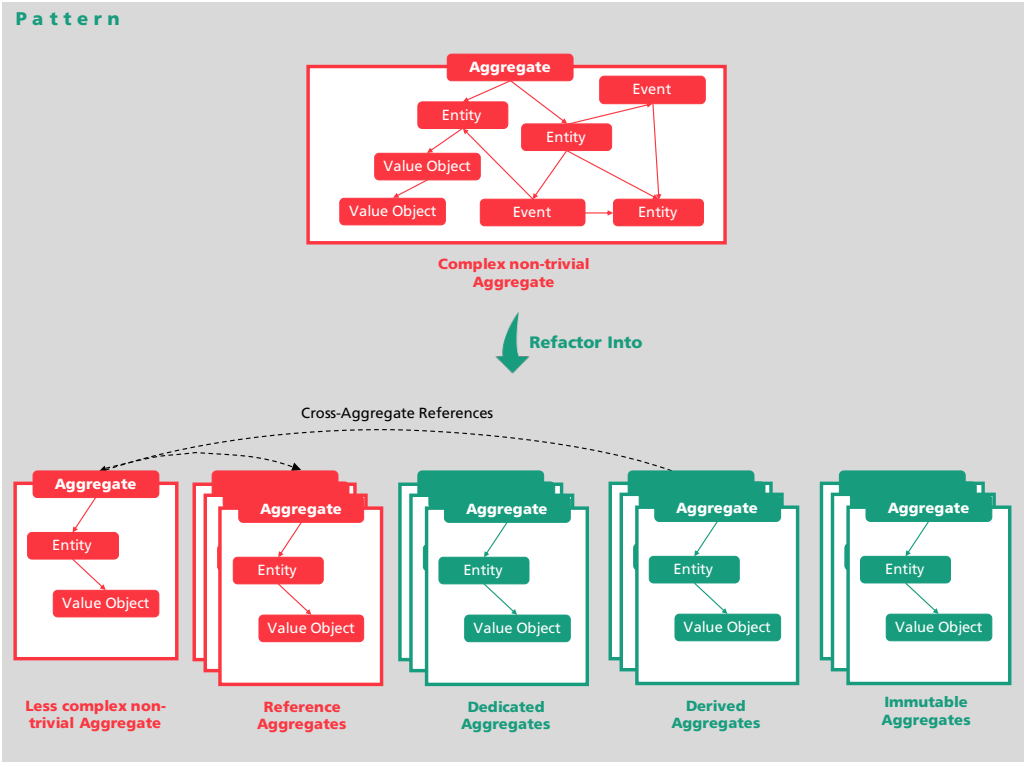

Figure 7.2: The Segregate Aggregate Classes Pattern

References

[ACHM11] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in bloom: a CALM and collected approach. In Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings, pages 249–260. www.cidrdb.org, 2011.

[ANB+95] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: Definitions, implementation, and programming. Distributed Comput., 9(1):37–49, 1995.

[BBG+95] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In Michael J. Carey and Donovan A. Schneider, editors, Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995, pages 1–10. ACM Press, 1995.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.

[Bra17] Susanne Braun. Semantics-driven optimistic data replication: Towards a framework supporting software architects and developers. In 2017 IEEE International Conference on Software Architecture Workshops, ICSA Workshops 2017, Gothenburg, Sweden, April 5-7, 2017, pages 236–241. IEEE Computer Society, 2017.

[Bre00] Eric A. Brewer. Towards robust distributed systems (abstract). In Gil Neiger, editor, Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA, page 7. ACM, 2000.

[Bre12] Eric Brewer. Cap twelve years later: How the "rules" have changed. Computer, 45(2):23–29, 2012.

[Bre17] Eric Brewer. Spanner, truetime and the cap theorem. https://research.google/pubs/pub45855.pdf, 2017.

[CDE+13] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. ACM Transactions on Computer Systems (TOCS), 31(3):1–22, 2013.

[Coc05] Alistair Cockburn. Hexagonal architecture. https://alistair.cockburn.us/hexagonal-architecture/, 2005.

[Eva04] Eric J. Evans. Domain-driven design - tackling complexity in the heart of software. Addison-Wesley, 2004.

[Eva14] Eric Evans. Domain-Driven Design Reference: Definitions and Pattern Summaries. Dog Ear Publishing, 2014.

[Fid87] Colin J Fidge. Timestamps in message-passing systems that preserve the partial ordering. 1987.

[Fow05a] Martin Fowler. Cqs. https://martinfowler.com/bliki/CommandQuerySeparation.html, 2005.

[Fow05b] Martin Fowler. Event sourcing. https://martinfowler.com/eaaDev/EventSourcing.html, 2005.

[Fow11] Martin Fowler. Cqrs. https://martinfowler.com/bliki/CQRS.html, 2011.

[GHOS96] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis E. Shasha. The dangers of replication and a solution. In H. V. Jagadish and Inderpal Singh Mumick, editors, Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996, pages 173–182. ACM Press, 1996.

[GR93] Jim Gray and Andreas Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993.

[GS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In Umeshwar Dayal and Irving L. Traiger, editors, Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987, pages 249–259. ACM Press, 1987.

[HA20] Joseph M. Hellerstein and Peter Alvaro. Keeping CALM: when distributed consistency is easy. Commun. ACM, 63(9):72–81, 2020.

[Hel10a] Joseph M. Hellerstein. Datalog redux: experience and conjecture. In Jan Paredaens and Dirk Van Gucht, editors, Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA, pages 1–2. ACM, 2010.

[Hel10b] Joseph M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. SIGMOD Rec., 39(1):5–19, 2010.

[Hel16] Pat Helland. Immutability changes everything. Commun. ACM, 59(1):64–70, 2016.

[Hoh04] Gregor Hohpe. Starbucks does not use two-phase commit. https://www.enterpriseintegrationpatterns.com/ramblings/18_starbucks.html, 2004.

[HR83] Theo Härder and Andreas Reuter. Principles of transaction-oriented database recovery. ACM Comput. Surv., 15(4):287–317, 1983.

[JPR+83] Douglas Stott Parker Jr., Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David A. Edwards, Stephen Kiser, and Charles S. Kline. Detection of mutual inconsistency in distributed systems. IEEE Trans. Software Eng., 9(3):240–247, 1983.

[Kle16] Martin Kleppmann. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly, 2016.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. Commun. ACM, 21(7):558–565, 1978.

[LLC] Google LLC. Google spanner documentation - replication. https://cloud.google.com/spanner/docs/replication?hl=en.

[Mar03] Robert C. Martin. Agile software development: principles, patterns, and practices. Prentice Hall PTR, 2003.

[Mat88] Friedemann Mattern. Virtual time and global states of distributed systems. 1988.

[Ric] C. Richardson. Microservices Patterns: With examples in Java. Manning Publications.

[RSS15] Erhard Rahm, Gunter Saake, and Kai-Uwe Sattler. Verteiltes und Paralleles Datenmanagement: Von verteilten Datenbanken zu Big Data und Cloud. Springer, 2015.

[TDP+94] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, USA, September 28-30, 1994, pages 140–149. IEEE Computer Society, 1994.

[Ter08] Douglas B. Terry. Replicated Data Management for Mobile Computing. Synthesis Lectures on Mobile and Pervasive Computing. Morgan & Claypool Publishers, 2008.

[Ver13] Vaughn Vernon. Implementing domain-driven design. Addison-Wesley, 2013.

[Vog09] Werner Vogels. Eventually consistent. Commun. ACM, 52(1):40–44, 2009.

[VV16] Paolo Viotti and Marko Vukolic. Consistency in non-transactional distributed storage systems. ACM Comput. Surv., 49(1):19:1–19:34, 2016.

[WVKG08] Ting Wang, Jochem Vonk, Benedikt Kratz, and Paul W. P. J. Grefen. A survey on the history of transaction management: from flat to grid transactions. Distributed Parallel Databases, 23(3):235–270, 2008.