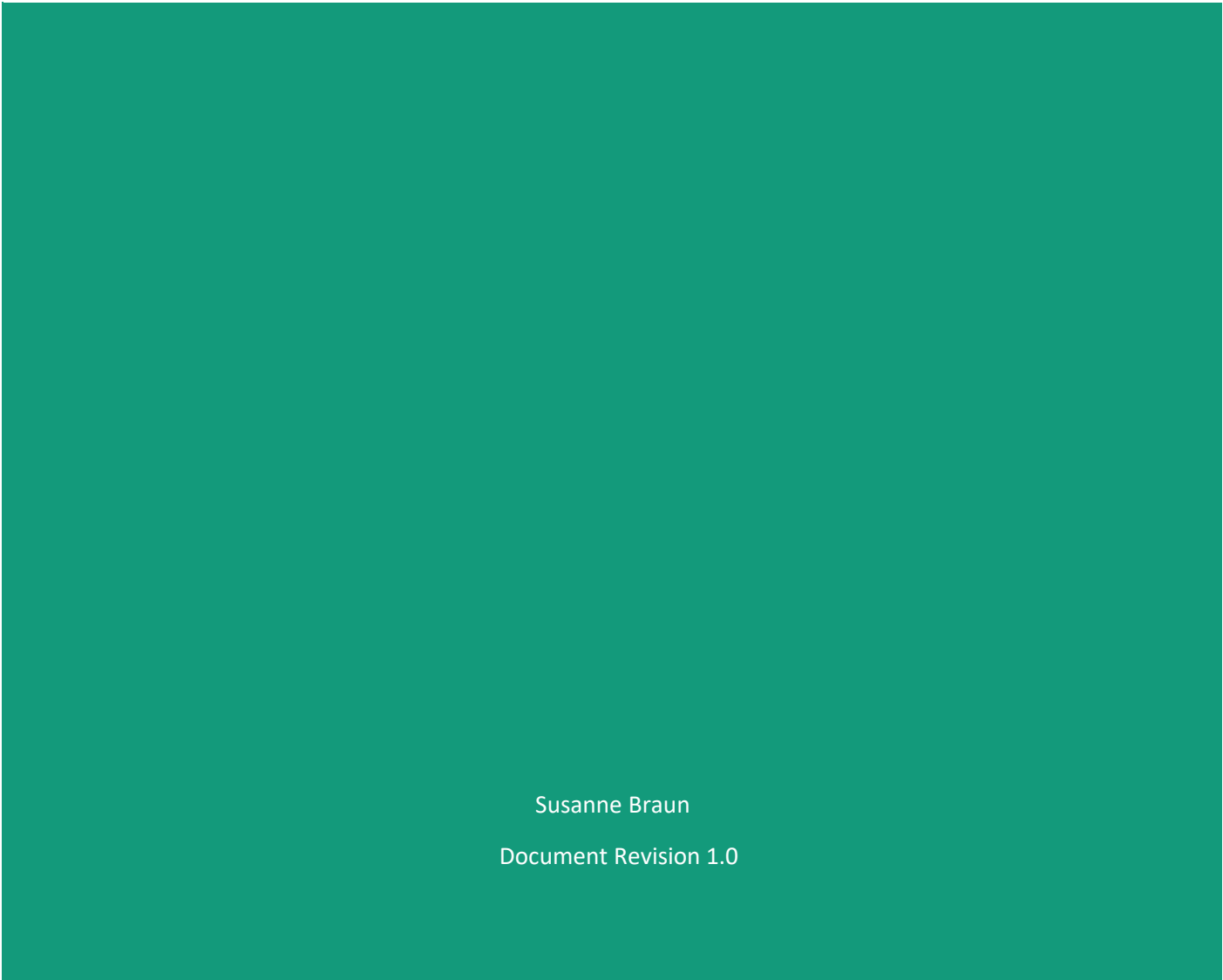




ECD3 Domain Operation Design Guide for Distributed Data-Intensive Systems



Susanne Braun
Document Revision 1.0

About this guide



[ECD3 Domain Operation Design Guide for Distributed Data-Intensive Systems](#) by [Susanne Braun](#) is licensed under [CC BY-SA 4.0](#) 

Document Revision: 1.0, February 2021

Inhaltsverzeichnis

1	WHAT IS THIS ALL ABOUT?	3
2	THE ECD3 REPLICATION FRAMEWORK	3
2.1	UPDATE PROPAGATION & RECONCILIATION IN ECD3	4
3	BEST PRACTICE: DESIGN FOR 'TOLERANCE TO PARTIAL EXECUTION ORDER'	5
4	COMPATIBILITY OF DOMAIN OPERATIONS	8
5	BEST PRACTICE: DESIGN FOR INCREMENTAL UPDATES	9
6	BEST PRACTICE: DESIGN FOR TRUE BLIND UPDATES	11
7	COMPATIBILITY ANTIPATTERNS.....	12
7.1	ANTIPATTERN: COARSE-GRAINED TECHNICAL BLIND UPDATES	13
7.1.1	HOW TO AVOID THIS?	15
7.2	ANTIPATTERN: IMPLEMENTING INCREMENTAL UPDATES AS BLIND UPDATES.....	15
7.2.1	HOW TO AVOID THIS?	17
7.3	ANTIPATTERN: POINTLESS USE OF APPEND-ONLY-STORAGE PRINCIPLES.....	18
7.3.1	HOW TO AVOID THIS?	18
8	BEST PRACTICE: DESIGN FOR DOMAIN INVARIANT CONSISTENCY.....	20
9	BEST PRACTICE: CONSIDER DURABILITY REQUIREMENTS.....	20
9.1	PATTERN: THE TENTATIVE OPERATIONS PATTERN	20
10	CHEAT SHEET	22
11	REFERENCES	23

ECD3 Domain Operation Design Guide for Distributed Data-Intensive Systems

1 What is this all about?

This is the ECD3 Domain Operation Design Guide. It will help you design domain logic for data-intensive systems in an optimized way and assist you in properly addressing concurrency-control-related design challenges appropriately. 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 layer to the domain layer, and you as a software architect and developer will have to deal with it. But let's first briefly recap what relational databases used to do in order to provide isolation guarantees in the context of transactions and what researchers propose to do.

In order to control the concurrent execution of read and write operations executed against relational databases, the latter consider **compatibility relationships** of read and write operations. Weikum et al. [1] propose Multilevel Transactions, a more generalized form of compatibility relationships that apply not only to read and write semantics but to business operations in general. They suggest commutativity as a criterion for compatibility. Helland proposes a similar concept with ACID 2.0 [2], but he demands business operations to be associative and idempotent in addition. However, designing your domain operations for commutativity plus associativity plus idempotence is hard to achieve in practice. In this guide, we provide different criteria for determining the compatibility relationships of domain operations. Keeping these criteria in mind will help you design your domain operations to be compatible, allowing operations to run concurrently and conflict-free on different replication nodes. In addition to an optimized design, all you need is an operation-based replication framework, as outlined in the next section.

2 The ECD3 Replication Framework

What distinguishes the ECD3 replication framework from most other replication frameworks is that it is an **operation-based replication framework with an operation-sending protocol**. It integrates with Domain-Driven Design [3] (DDD). DDD decomposes the application domain into subdomains. Therefore, ECD3 does not consider databases as the units of replication, but **subdomain models**. Figure 1 shows a standard three-layer architecture where Domain-Driven Design is applied [4]. The top layer, the application layer, hosts the coarse-grained business operations that usually span complete user stories. Code-wise, these are implemented using methods of an application service protected by transaction boundaries. Using the dependency inversion principle (short: DIP) [5], transactions are started and ended with the start and the termination of the application service method. Application service methods are

the client of the subdomain models hosted in the domain layer. Domain operations of subdomain models are usually implemented in aggregates or domain services. In principle, we distinguish pure read operations (query methods) and updating domain operations (update methods) in the domain model. The domain operations in turn use the infrastructure layer to persist subdomain models. In the infrastructure layer, read and write operations are executed against data items persisted in the database.

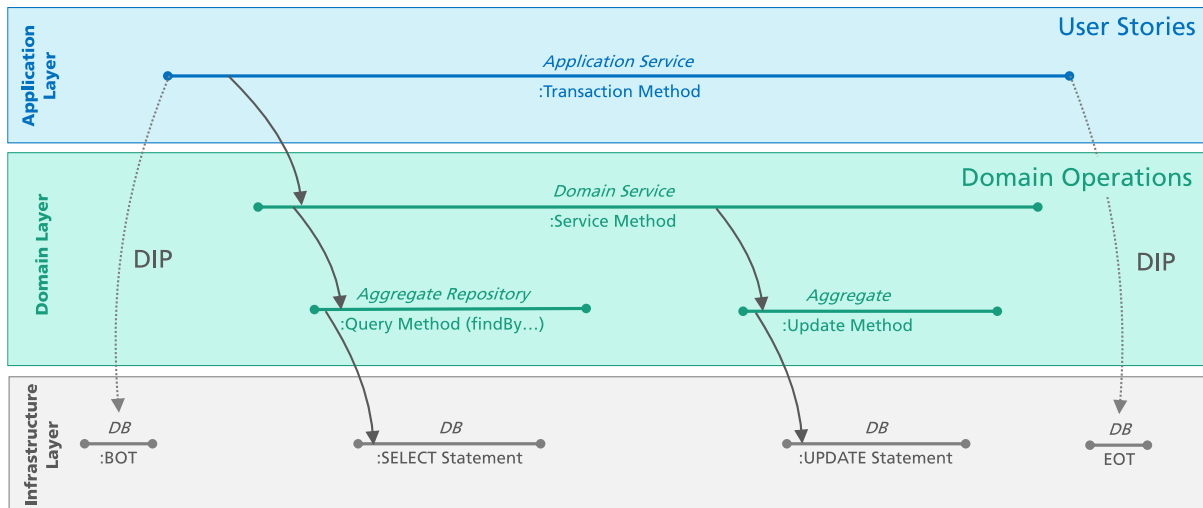


Figure 1 Domain operations in Domain-Driven Design

2.1 Update Propagation & Reconciliation in ECD3

In the following, we will use an example subdomain model from the banking domain to illustrate how updates are propagated and reconciled with other concurrent updates by the ECD3 framework.

Our sample subdomain model contains an aggregate “Account” with the corresponding root entity. The root entity has a property “balance” and the domain operations withdraw “w(amount)” and deposit “d(amount)”. Replica₁ replicates this subdomain model. The subdomain model holds three account instances: accounts a, b, and c. At some time, Replica₁ exchanges updates with other replicas and learns that there have been concurrent domain operation executions at Replica₂. A transaction at Replica₂ transferred 1000 € from account c to account a, while at Replica₁, a transaction transferred 500 € from account a to account b. ECD3 replicas maintain an operation log that is organized as a graph as illustrated in Figure 2. Each branch of the graph corresponds to the operation execution history of one replica.

If all concurrent operations are compatible (withdraw and deposit are commutative operations), the concurrent operations can be reconciled by ECD3 with the following reconciliation algorithm:

- Create a new branch for reconciliation
- Copy all operations of the concurrent branches to the reconciliation branch
- Sort operations deterministically on the reconciliation branch so that happens-before-relationships of operations are preserved (version vectors [40] can be used for that)
- Re-execute all operations on the reconciliation branch on a copy of the domain model snapshot that was valid before the replicas diverged
- Persist the new snapshot with the updates executed on the reconciliation branch

- Switch to the reconciliation branch for further execution

After reconciliation, Replica₁ will run its local transactions on the reconciliation branch. The old branch will be marked as terminated, as illustrated in Figure 3.

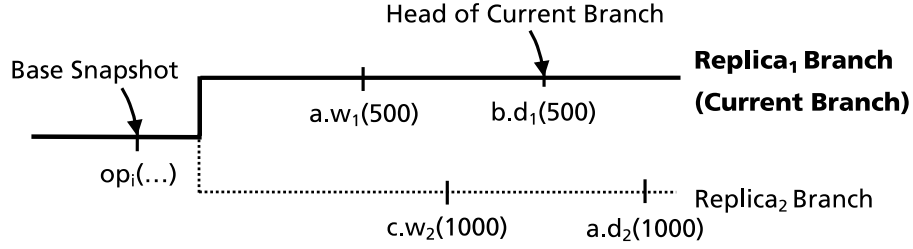


Figure 2 Operation log at Replica₁ after update exchange

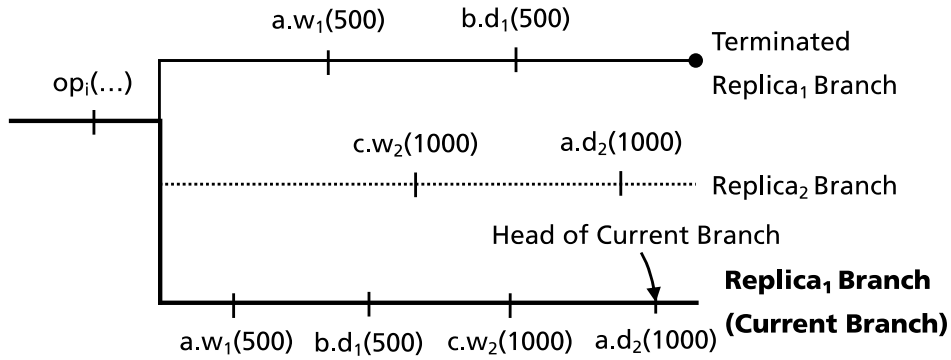


Figure 3 Operation log at Replica₁ after reconciliation

By sorting operations according to their happens-before-relationships, ECD3 guarantees causal consistency. The central idea of causal consistency is to preserve causal dependencies between different updates so that updates with causal dependencies are always observed in the proper order by application programs. A causally consistent replicated system ensures that if update u_2 has been executed after u_1 , then u_2 is executed after u_1 on all other replicas as well [14]. For example, a comment on a task description in a task management system should only be observed if the task description can be observed as well. Note that we provide a deterministic algorithm for sorting operations on the reconciliation branch so that eventually all operations will be re-executed in the same execution order on all replicas. Thus, strictly speaking, the ECD3 framework guarantees sequential consistency, which is even stronger than causal consistency [8].

3 Best Practice: Design for ‘Tolerance to Partial Execution Order’

Updating domain operations of replicated subdomain models need to fulfill one important property: They need to be **tolerant to partial ordering of the execution order**.

We will use an example from the task management subdomain (part of a remote work application domain) to illustrate the meaning of this important property: The entity Task has an attribute description, which is a mutable value object of the type Text. In this simplified example, Text has only one domain operation $i(\text{int pos}, \text{Char c})$ for the insertion of characters into the text buffer in a specific position. An excerpt of the subdomain model is given in Figure 4.

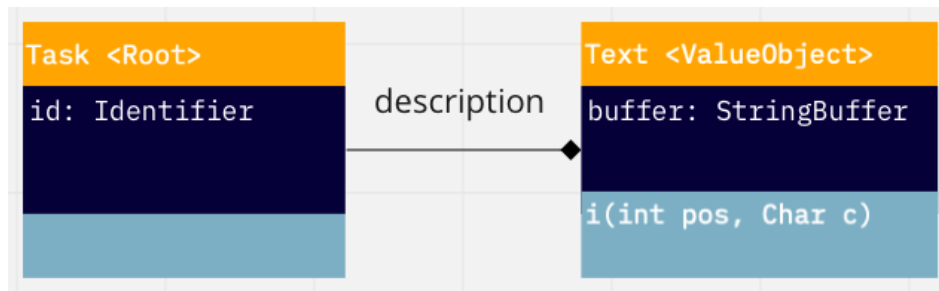


Figure 4 'Task' entity and 'Text' value object from the task management subdomain

Assume we have one instance of Text t with a current value of $t.buffer$ "Hlo Wold". The instance is replicated on Replica₁ and Replica₂. Figure 5 shows the execution history from the viewpoint of Replica₁. Replica₁ executed two inserts to fix spelling errors in the first word: "Hlo Wold" became "Hello Wold". At Replica₂, another insert was executed to fix the spelling error in the second word: "Hlo Wold" became "Hlo World". But, as you can see, the re-execution of $t.i_2(6, 'r')$ on the reconciliation branch is unable to produce the originally intended value change of the original execution. Because of other concurrent operations at Replica₁, the operation of Replica₂ is executed on a different state than during the original execution. The position parameter then points to a wrong place. This could be fixed with immutable text positions that allow insertion of new immutable text positions at runtime, such as DeweyIDs [42].

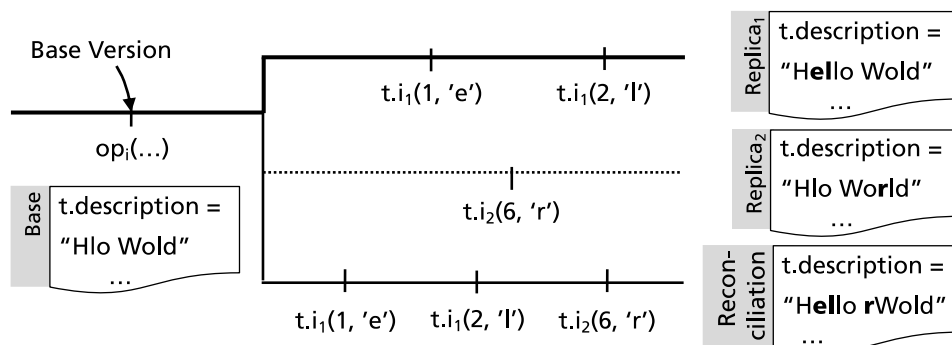


Figure 5 Updates without partial ordering tolerance

The problem is that there is no happens-before-relationship between concurrent operations. Therefore, there is no unique ordering of concurrent operations according to causal consistency. Causal consistency provides only a partial ordering of domain operations and can only provide ordering guarantees for non-concurrent operations. In the example, domain operation i was not designed to be tolerant to the partial ordering of causal consistency. As already mentioned, immutable text positions could be used to achieve this property for domain operation i .

We will now provide a more formal definition of the tolerance-to-partial-ordering property of domain operations. For this purpose, we consider only updating domain operations and view them as operations that (1) reside in the domain layer, (2) are invoked by application service methods in the context of transactions, and (3) perform a number of updates on different object attributes of various aggregates. This view on updating domain operations is illustrated in Figure 6. Updating domain operations can be implemented as methods of domain services, aggregates, or domain objects. Domain objects can be entities or value objects.

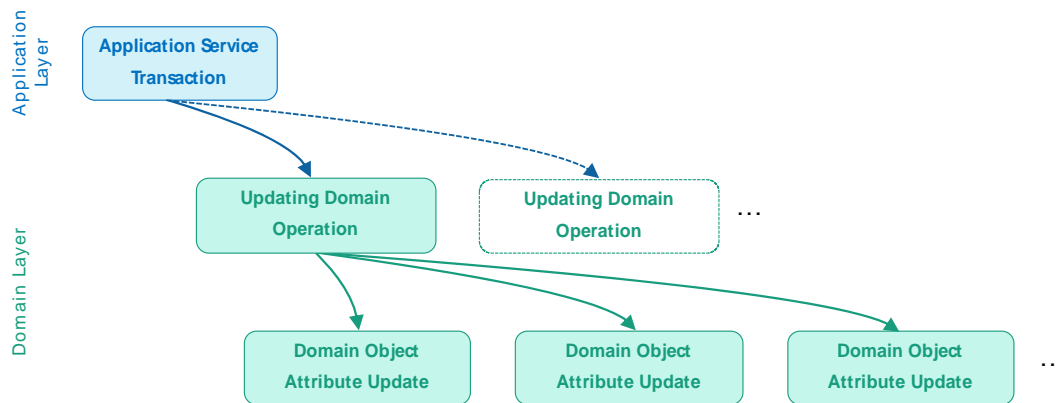


Figure 6 Updating domain operations in ECD3

Domain-Driven Design considers the following domain object attribute types:

- primitive values
- references to value objects
- references to entities
- references to other aggregates (by ID of the root entity of the referenced aggregate)

Therefore, an attribute update can be defined as follows:

Def.: Attribute Update

A domain operation performs an **attribute update** if, as a result of the domain operation execution, the observable state of a domain object attribute has changed. Depending on the attribute type, the observable state of a domain object attribute is changed if:

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

Instead of “domain object attribute update”, we will use “attribute update” for short in the following. As domain operations can perform different attribute updates, a domain operation can only be tolerant to partial ordering of the execution order if all attribute updates performed by the operation fulfill this criterion. This leads us to our next definition:

Def.: Partial Ordering Tolerance of a Domain Operation

A domain operation is tolerant to partial ordering of the execution order if all attribute updates performed by the operation are tolerant to partial ordering.

And finally:

Def.: Partial Ordering Tolerance of an Attribute Update

Let

1. attribute update u be concurrent with attribute updates c_1, \dots, c_n
2. c_1, \dots, c_n be an execution ordering that respects all happens-before relationships of c_1, \dots, c_n
3. u , as well as c_1, \dots, c_n update the same attribute $o.att$ of some domain object o

Attribute update u is **tolerant to partial ordering of the execution order** if u can be re-executed on the reconciliation branch after c_1, \dots, c_n and still correctly produces **the same intended attribute changes** as during the original execution.

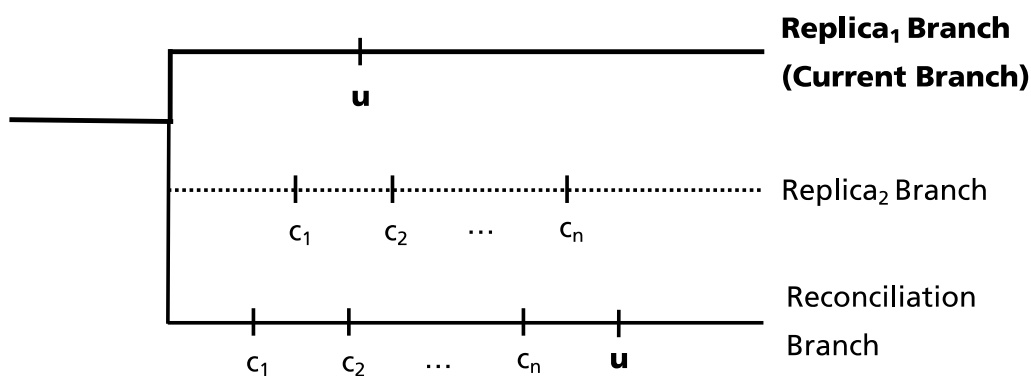


Figure 7 Partial ordering tolerance of attribute updates

Of course, the meaning of “intended attribute change” highly depends on the domain semantics and needs to be decided individually. We will provide more examples in the course of this guide. **In essence, a domain operation fulfills the criteria of “partial ordering tolerance” if – under the assumption that happens-before-relationships are preserved – the domain operation can be re-executed on a different state of the subdomain model that contains attribute updates of concurrent operations.** The partial ordering tolerance of domain operations is an important quality and a prerequisite for establishing compatibility relationships between domain operations, which we will describe in the following.

4 Compatibility of Domain Operations

In order to determine the compatibility of two domain operations, we use the following two criteria, which must be fulfilled:

1. Both operations need to be tolerant to partial ordering of the execution order.
2. Both operations do not overwrite each other’s changes (except for true blind updates) during reconciliation.

It is obvious that operations with disjoint update sets (operations that update different domain objects or different attributes of the same domain objects) do not overwrite each other’s changes. But what if two domain operations potentially update the same attributes of the same domain object instances? How can we design these operations in such a way that

they do not overwrite each other's changes? We will explain this in the following sections by means of best practices and antipatterns and also illustrate the meaning of true blind updates.

5 Best Practice: Design for Incremental Updates

One possibility to ensure that domain operations do not overwrite each other's changes is to design them in such a way that attribute updates are executed as "incremental updates", as defined below:

Def.: Incremental Update

An **incremental update** is an attribute update whose outcome is dependent on the current state of the attribute to be updated. Therefore, prior to updating the attribute, the domain operation has to **evaluate** the current state of the attribute in order to **derive** the updated attribute value.

Thus, the schedule of such a domain operation must contain read access to the attribute **prior** to write access, as illustrated in Figure 8.

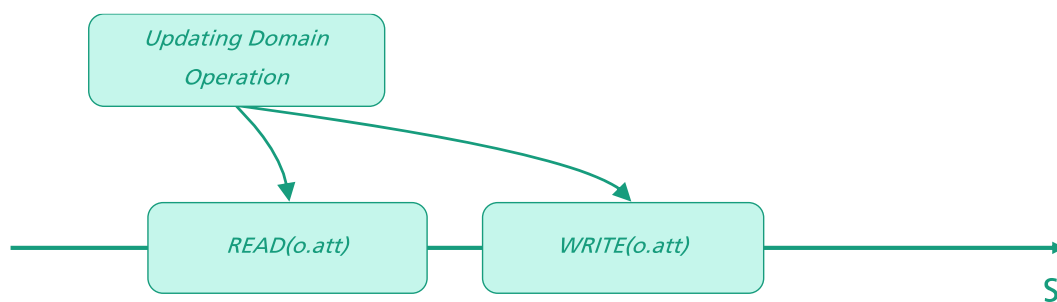


Figure 8 Incremental attribute update of a domain object attribute "o.att"

Examples

Example 1: Counter Increments

Updating a counter value attribute by adding an increment value is an incremental operation as the current value of the counter has to be evaluated first in order to calculate the incremented value.

```

class Game {
    int points;
    ...

    void finalBossDefeated() {
        points = points + 100;
    }

    ...
}

```

Listing 1 Counter increment in pseudocode

Example 2: List Appends

Updating a list attribute by adding a new element at the end of the list requires evaluating the list first in order to increase the list size by one and insert the new element at position (size-1).

```

class Playlist {
    List<Identifier> songs = new ArrayList<>();

    ...

    void appendSongAtEnd(Identifier songId) {
        songs.add(songId);
    }

    ...
}

```

Listing 2 List append in pseudocode



- As domain operations explicitly have to consider the current state of the attribute in order to perform an incremental update, such updates cannot overwrite the changes of concurrent operations executed prior on the reconciliation branch.
- **But:** As incremental updates explicitly consider the current state, **special care has to be taken to ensure that incremental updates are tolerant to partial ordering of the execution order.** Revisit the collaborative text editing example of the previous section, which demonstrates this challenge with an incremental update on the `Text.buffer` attribute. The examples given in this section – counter increments and list appends – are tolerant to partial ordering of the execution order.

6 Best Practice: Design for True Blind Updates

It is okay for domain operations to overwrite changes of concurrent operations if the updates are **true** blind updates, as defined below:

Def.: True Blind Update

A **true blind update** is kind of the opposite of an incremental update: The attribute is updated by the domain operation to an updated state that is completely independent of the current state of the attribute. One could say that the attribute is “blindly overwritten” by the domain operation “in any case” and without the need to evaluate it first.

Thus, syntactically the schedule of a domain operation performing a true blind update on some attribute ***o.att*** does **not** contain read access to the attribute prior to write access, as illustrated in Figure 9.

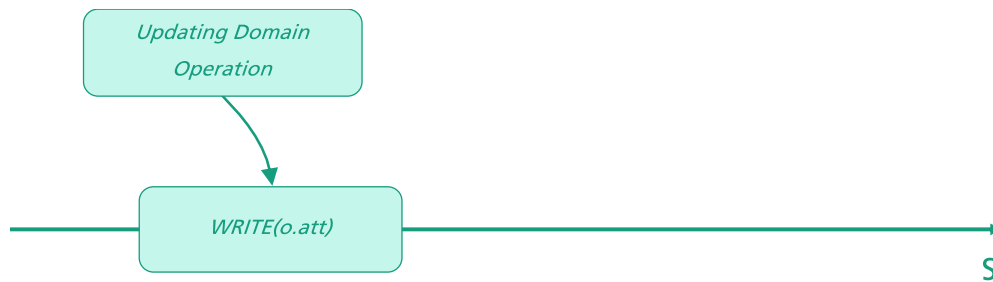


Figure 9 Syntactic schedule of a true blind update

Note that it might still be the case that the domain operation is implemented in such a way that it also reads the value of the attribute before writing it. But the important thing is that the read value of the attribute does not have an impact on the final value written to the attribute.

Examples

Example 1: Counter Resets

Resetting counters are often true blind updates as counter resets are quite often triggered by the occurrence of certain domain events that are independent of the current counter value. Of course, this depends on the concrete domain semantics. But consider an online game as an example. When players decide to restart the game, their current high score counters are reset in any case, independent of the current value of the counters.

```

class Game {
    int points;

    ...
    void restart() {
        points = 0;
    }
}

```

Listing 3 Counter reset in pseudocode

Example 2: List Clearings

Similar to counter resets, list clearings are quite often true blind updates. Consider a user's watchlist at a video streaming provider. If the user decides to completely empty this list in order to restart with an empty list that they can repopulate with movies matching their current taste, then the list can be emptied independent of its current contents.

```

class Playlist {
    List<Identifier> songs = new ArrayList<>();

    void deleteAll() {
        songs.clear();
    }
}

```

Listing 4 List clearing in pseudocode



- True blind updates blindly overwrite changes of concurrent operations executed before on the reconciliation branch. But this is okay, as these updates would have been executed in any case independent of the current value of the attributes and therefore also independent of any changes concurrent operations might have performed on these attributes before.
- True blind updates are always tolerant to partial ordering of the execution order as blind updates blindly overwrite the current state of an attribute independent of its current state. Thus, any attribute updates of concurrent operations executed before on the reconciliation branch are also simply overwritten and the attribute is updated to the defined value.

7 Compatibility Antipatterns

In the following, we will provide some antipatterns that undermine the compatibility of domain operations.

7.1 Antipattern: Coarse-Grained Technical Blind Updates

An antipattern often used in combination with state-based update propagation are “technical blind updates”. Technical blind updates undermine the compatibility of domain operations in designs with operation-sending protocols and lead to unnecessary lost updates in case of high concurrency. A domain operation performs technical blind updates if it blindly writes all attributes of a domain object (or aggregate), instead of writing only attributes that have changed as a result of some real domain logic computation or because the user provided updated values. You can quite often observe this kind of antipattern when the state is propagated in data transfer objects (DTO). The DTOs are then passed to the domain operation, which processes them as illustrated in Listing 5. In this example, all attributes of a Profile are blindly written even if only a single attribute or a selected number of attributes have changed in comparison to the current state of the profile.

```
class UserProfileService {  
    UserProfileRepository repo;  
  
    ...  
  
    void updateUserProfile(UserProfileDTO profileDTO) {  
        UserProfile profile = repo.getProfileById(profileDTO.getId());  
  
        profile.setFirstName(profileDTO.getFirstName());  
        profile.setLastName(profileDTO.getLastName());  
        ...  
        profile.setEmail(profileDTO.getEmail());  
  
        repo.update(profile);  
    }  
}
```

Listing 5 Technical blind updates with DTOs

Note that refactoring the code so that an attribute is only updated if its current values are different from the values transmitted in the DTO makes it worse. As you can see in Listing 6, the attribute updates are then no longer blind updates, as the current values of the attributes are evaluated first in order to make the comparison with the values of the DTO. As a result, we now also need to take care of the tolerance to partial execution order guarantee. Unfortunately, this implementation of the operation is not tolerant to partial ordering. The issue is illustrated in Figure 10: In the sample operation log, an unnecessary lost update occurs because of this design flaw. The update on the email attribute originally executed at Replica₂ is overwritten by an unintended attribute change happening during the re-execution of the updateProfile operation from Replica₁.

```

class ProfileService {

    ProfileRepository repo;

    ...

    void updateProfile(ProfileDTO profileDTO) {
        Profile profile = repo.getProfileById(profileDTO.getId());

        if(profile.getFirstName() == null ||
            !profile.getFirstName().equals(profileDTO.getFirstName())) {
            profile.setFirstName(profileDTO.getFirstName());
        }
        If(profile.getLastName() == null ||
            !profile.getLastName().equals(profileDTO.getLastName())) {
            ...
        }

        repo.update(profile);
    }
}

```

Listing 6 Sample operation design that is not tolerant to partial ordering of execution order.

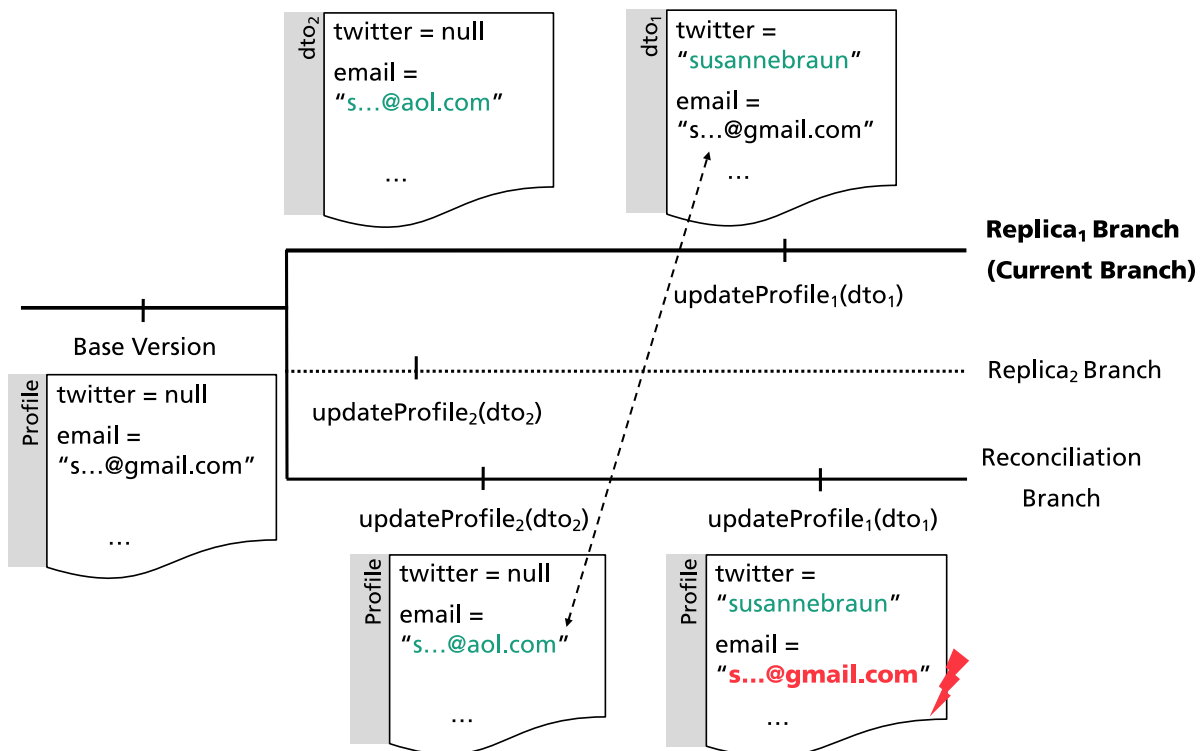


Figure 10 Operation log with operations not tolerant to partial execution order (originally intended attribute updates are given in green font)

7.1.1 How to avoid this?

- Define fine-granular updating domain operations that update only single attributes or that perform few coherent attribute updates. Invoke only those domain operations that perform required state changes on attributes.
- Design the domain operations in your subdomain models to execute real domain logic and avoid anemic models containing only getters, setters, and generic “updateDomainObject(dto)” operations.
- If you still need to implement generic “updateDomainObject(dto)” operations: Set attributes without state changes to null in the DTO and ignore null-valued attributes when applying the updates to your corresponding domain objects.

7.2 Antipattern: Implementing Incremental Updates as Blind Updates

Another antipattern leading to unnecessary lost updates is the implementation of updates that are, from the business perspective (and/or UX design), inherently incremental updates as blind updates.

Consider a cloud service of a remote work platform that realizes a mood barometer service for remote teams. Each remote coworker has the same fixed budget of “mood points”. Remote coworkers can share their current mood with the team by spending “mood points” in order to “upvote” moods from a set of predefined moods such as “productive”, “focused”, “stressed”, and so on. Spent points can be freed anytime by downvoting moods that have previously been upvoted. Freed points can then in turn be used again to upvote other moods and so on. The mood barometer visualizes an aggregated and anonymized view of the current distribution of mood points spent by the members of the team. The team mood visualization is given in Figure 11. Thus, the mood barometer service realizes a live indicator for the current team mood.



Figure 11 Mood visualization of a team barometer

Now spending and freeing mood points for the moods of the barometer is an inherently incremental operation like counter increments or decrements. For example, the corresponding upvote domain operation for spending points can be implemented as given in Listing 7. The related data model design is given in Figure 11.



Figure 12 Team mood barometer data model

```

class ImmutableMoodBarometer {

    TeamMemberVotesRepository votesRepo;
    ...

    void upVote(Identifier voterId, Identifier moodId) {
        TeamMemberVotes votes = votesRepo.
            getByVoterIdAndMoodBarometerId(voterId, this.id);
        Map<Identifier, Integer> votesByMoodId = votes.getVotesByMoodId();

        // invariant check
        if(votes.getTotalNumberOfSpentPoints() >=
            config.getMaxVotesPerUser()) {
            throw new RuntimeException("Maximum number of points spent!");
        }

        // spending a point
        Integer updatedMoodPoints = 1;
        if(votesByMoodId.get(moodId) != null) {
            updatedMoodPoints = votesByMoodId.get(moodId)+1; Attribute Read
        }
        votesByMoodId.set(moodId, updatedMoodPoints); Attribute Write
    }

    ...
}

```

Listing 7 Operation implementing incremental updates for spending mood points

An antipattern would be to implement the upvote() and downvote() operations with blind updates by replacing them with an updateTeamMemberVotes(teamMemberVotesDto) operation as given in Listing 8.

As you can see in Figure 13 and Figure 14, concurrent executions of operations realizing incremental updates produce the correct intended attribute changes during reconciliation (Figure 13), while operations implementing blind updates lead to lost updates during reconciliation (Figure 14). Note that for the sake of readability, we used self-explanatory mood identifiers instead of technical mood IDs in the graphics.

7.2.1 How to avoid this?

- Design and implement updates that are already naturally incremental from the business perspective as incremental updates and not as blind updates.

```

class ImmutableMoodBarometer {
    TeamMemberVotesRepository votesRepo;

    ...

    void updateTeamMemberVotes(teamMemberVotesDto dto) {
        TeamMemberVotes votes = votesRepo.
            getByVoterIdAndMoodBarometerId(dto.getVoterId(), this.id);

        // blindly overwrite all spent points
        votes.setVotesByMoodId(dto.getVotesByMoodId());    Attribute Write

        // invariant check
        if(votes.getTotalNumberOfSpentPoints() >
            config.getMaxVotesPerUser()) {
            throw new RuntimeException("Maximum number of points spent!");
        }
    }

    ...
}

```

Listing 8 Operation implementing blind updates for spending and freeing mood points

7.3 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 it 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 still has observable state changes and thus is 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 result in higher compatibility of domain operations, 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.

7.3.1 How to avoid this?

- Do not mix up true immutability of aggregates indicated from the business semantics with technical concepts developed for optimizing write access times for special types of data, such as time-series data in append-only stores.

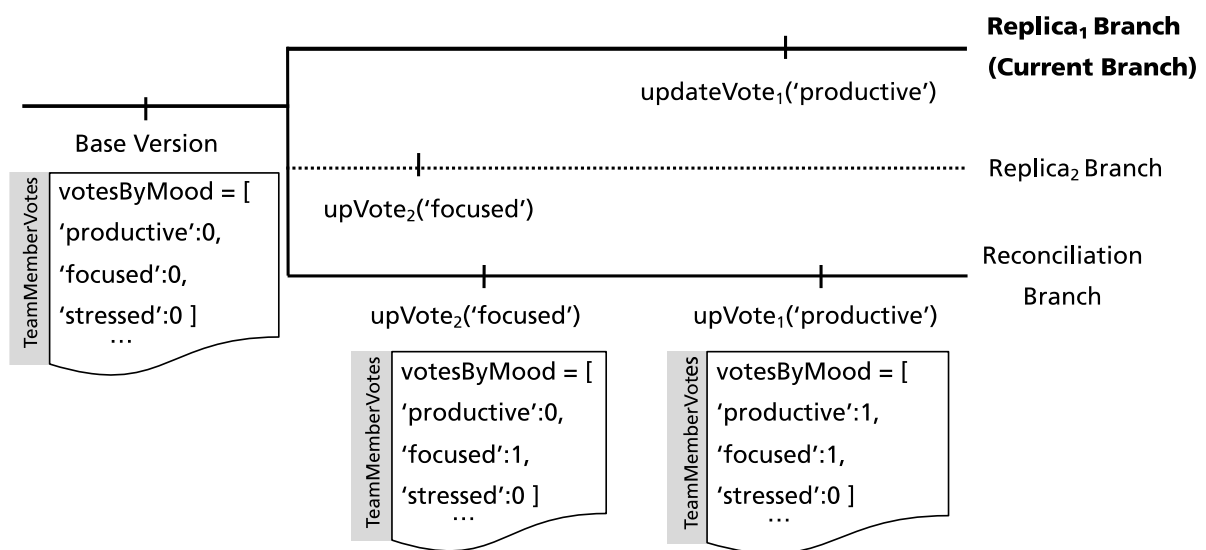


Figure 13 Incremental Updates produce intended attribute changes during reconciliation

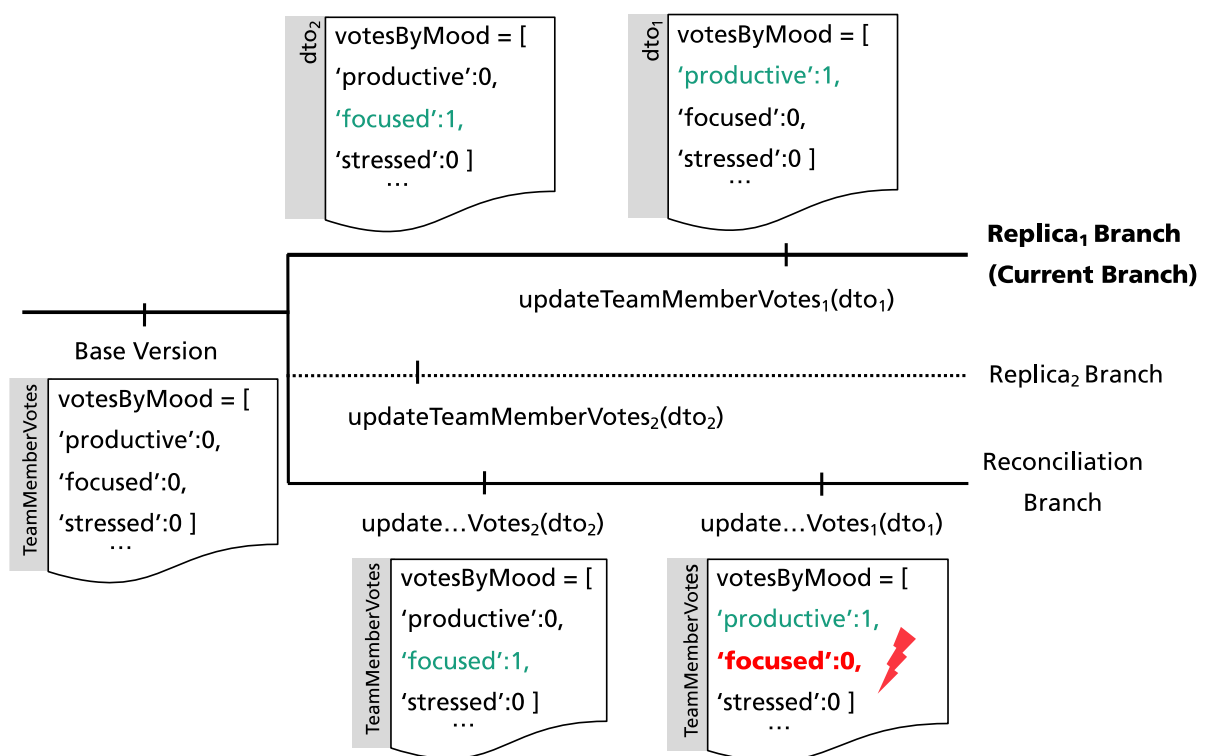


Figure 14 Incremental updates implemented as blind updates lead to lost updates during reconciliation


8 Best Practice: Design for Domain Invariant Consistency

Software architects and developers quite often deal with domain invariants, which are usually business-critical. Therefore, as a minimum requirement, the ECD3 framework assumes that domain operations are implemented correctly so that all domain invariants, in particular, are kept. To avoid violation of domain invariants, a lot of domain operations first read some attributes in order to evaluate the invariant condition. Depending on the outcome of the evaluation, attribute updates are either performed or not.

Examples

For example, in a banking application, a domain invariant might be that the balance of a salary account must always be positive.

Examples:



Banking – Withdraw


```
withdraw(amount) {  
    ...  
     assert(balance > dispoLimit)  
}
```

Figure 15 Example of an invariant-preserving operation from the banking domain

9 Best Practice: Consider Durability Requirements

In addition to compatibility relationships, ECD3 also considers durability requirements of domain operations. Unfortunately, with eventual consistency it might turn out in retrospective that, due to other concurrent operations, the re-execution of a domain operation during reconciliation may lead to a domain invariant violation. In this case, the operation has to be aborted during reconciliation and its effects on the original subdomain model replica are not durable.

9.1 Pattern: The Tentative Operations Pattern

If a domain operation does require durability, it should be implemented as a “tentative” domain operation in analogy to tentative transactions, as already proposed in [9]. Tentative operations are marked as tentative as long as the operation has not been successfully executed at a so-called master replica. As soon as it has been successfully executed at the master, it becomes durable. Thus, if an aggregate has tentative operations, it requires a dedicated replica that globally takes care of serializing access to that aggregate.



- Please note that non-tentative operations are not guaranteed to be durable, even if the operations do not affect domain invariants. Durability cannot be guaranteed, as during re-execution, an operation can always take a different program flow than on the originating replica, resulting in a different outcome. This is especially true when domain operations are re-executed on a snapshot containing updates of concurrent operations.

Examples

Domain operations that update the availability status of a limited resource, like the updating of the booking status of a room or an airplane seat.

10 Cheat Sheet

If you design your domain operations to be **tolerant to partial execution order** and implement all attribute updates of your domain operations either as

1. **incremental updates** or
2. **true blind updates**

your domain operations will be **compatible** with concurrent execution under eventual consistency. Of course, you will also need an operation-based replication framework as described in section 2.

Therefore, when designing your domain operations, consider the following design best practices:

1. Design operations to be **tolerant to partial execution order** (section: Best Practice: Design for 'Tolerance to Partial Execution Order')
2. Design operations to perform **incremental updates** when business-wise and technically feasible (section: Best Practice: Design for 'Tolerance to Partial Execution Order')
3. Design operations to perform only **true blind updates** (section: Best Practice: Design for True Blind Updates)
4. **Do not forget business semantics and the user's intents**: Is the operation naturally incremental from this viewpoint or does it have the characteristic of a true blind update? (section: Antipattern: Implementing Incremental Updates as Blind Updates)
5. Avoid coarse-grained blind updates and anemic domain models (section: Antipattern: Coarse-Grained Technical Blind Updates)
6. Beware of your **domain invariants** and reason about an invariant's impact on **durability requirements** (section: Best Practice: Design for Domain Invariant Consistency)

11 References

- [1] G. Weikum und H.-J. Schek, "Concepts and applications of multilevel transactions and open nested transactions," in *Database transaction models for advanced applications*, San Francisco, Morgan Kaufmann Publishers Inc., 1992, pp. 515-553.
- [2] P. Helland und D. Campbell, "Building on Quicksand," in *Fourth Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, USA, 2009.
- [3] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2003.
- [4] V. Vaughn, *Implementing Domain-Driven Design*, Addison-Wesley, 2013.
- [5] R. C. Martin, *Agile software development: principles, patterns, and practices*, Prentice Hall, 2002.
- [6] C. Baquero und N. Preguiça, "Why Logical Clocks are Easy," *Communications of the ACM*, Vol. 59, No. 4, pp. 43-47, 2016.
- [7] D. B. Terry, *Replicated Data Management for Mobile Computing*, Morgan & Claypool Publishers, 2008.
- [8] M. P. Haustein, T. Härder, C. Mathis und M. Wagner, "DeweyIDs-The Key to Fine-Grained Management of XML Documents," in *SBBD*, 2005.
- [9] J. Gray, P. Helland, P. O'Neil und D. Sasha, "The dangers of replication and a solution," *ACM SIGMOD Record*, Vpl. 25, No. 2, pp. 173-182, 1996.
- [10] P. Helland, "Immutability Changes Everything," *ACM Queue - Structured Data*, Vol. 13, No. 9, p. 40, 2015.