In the domain-driven design (DDD) structure, aggregation is difficult to understand and is also a key obstacle in the DDD learning curve. Reasonably designed aggregation clearly expresses business consistency and is more likely to lead to clear implementations. However, poorly designed aggregation or the absence of aggregation in the design fails to lead to clear implementations.

The concept of aggregation is not complex. This article is intended to return to the essence of aggregation and give some valuable suggestions on its definition and practice.

# 1. Core Problems Solved by Aggregation

First, let's take a look at the definition of aggregation in the DDD Reference:

*Entity and value objects are divided into aggregation and define the boundary around the aggregation. Select an entity as the root of each aggregation and allow only external objects to hold references to the aggregation root. Define the attributes and invariants of the aggregation as a whole and assign responsibility for their execution to the aggregation root or the specified framework mechanism.*

This is typical pattern language that explains what aggregation is, what the aggregation root is, and how to use aggregations. However, the problem with the pattern language is over-refinement. It is easy to read if the reader is already familiar with the pattern but difficult for those that need to see it most or are not familiar enough with the concepts yet. To deeply understand the nature of a pattern, we still have to return to the core issues it tries to solve.

There is a famous saying in the field of software architecture:

"*The architecture is not determined by the function of the system but by the non-functional attributes of the system.*"

The straightforward explanation of this sentence is that if you do not consider performance, robustness, portability, modifiability, development cost, time constraints, and other factors, the functions of the system can always be realized, and the project can always be completed with any architecture and any method. It's just that the development time, future maintenance costs, and ease of feature expansion are different.

Of course, the reality is not so. We always hope the system can perform well in terms of understanding, maintenance, and scalability to achieve the business goals behind the system more quickly and economically. However, in reality, unreasonable design methods may increase the complexity of the system. Let's take a look at an example:

**Let's say the problem area is an office supplies purchasing system within an enterprise.**

- Employees of the enterprise can submit a purchase requisition through the system. A request contains several quantities and types of office supplies, namely purchase items. (1)

- The supervisor is responsible for approving procurement applications. (2)
- After the approval, the system will generate several orders according to different providers. (3)

For the same problem, there are several different design ideas, such as database-centered design, object-oriented design, and "correct OO" DDD design.

If a database-centered modeling method is adopted, the database design will be carried out first. Many teams are still taking this method and spending a lot of time discussing the database structure. We only give the forms related to the purchase requisition to avoid having a large chart. The architecture is shown in the figure below:

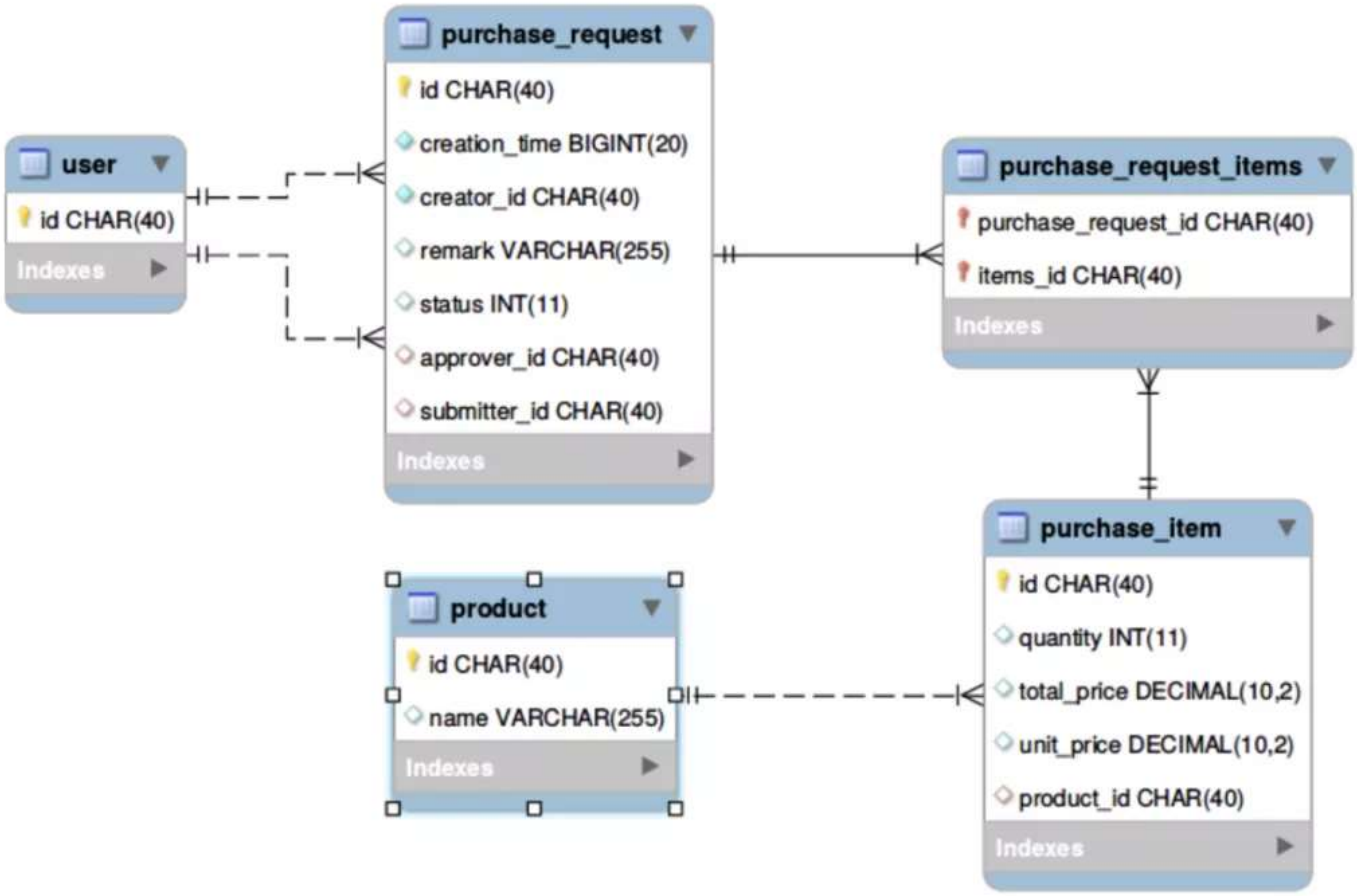


*Figure 1 – Design from the Perspective of the Database*

If the problem is considered directly at such a low design level of the database, in addition to the cumbersome and error-prone design of the database, more importantly, some more complex business rules and data consistency assurance will be faced. For example:

- If the purchase requisition is deleted, the corresponding purchase items related to the purchase requisition and their associations need to be deleted. In the database design, these constraints can be guaranteed by database foreign keys.

- Complex locking mechanisms may be involved if multiple users are processing relevant data concurrently. For example, if the approver is approving the purchase requisition and the purchase submitter is modifying the purchase item, it may cause the audited data to be stale data or cause the failure of the update of the purchase item.
- If certain associated data are updated at the same time, partial updates may also cause problems. In database design, these constraints are ensured by transactions.

Generally, every problem has a solution. However, first, the discussion of the model has prematurely entered the implementation, separated from the business concept, and is not convenient for continuous cooperation with business personnel. Second, the details of technology and the business rules are intertwined, so it is easy to be in an imbalanced condition. *Is there a solution that would allow more focus on the problem area rather than plunging deeper into such technical details?*

Object-oriented technology and object-relational mapping (ORM) help improve the abstraction level of the problem. In the object-oriented scenario, the structure is like this:

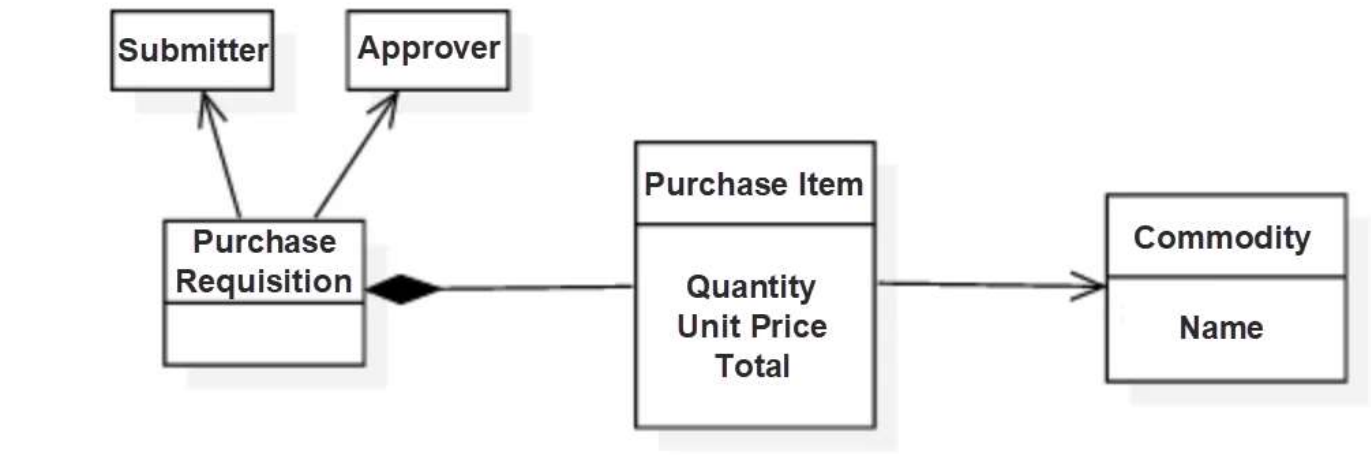


*Figure 2 – Design from the Perspective of Traditional OO*

The object-oriented method increases the level of abstraction and ignores unnecessary technical details. For example, we no longer need to care about technical details, such as foreign keys and associated tables. The number of model elements we need to care about is reduced, and the complexity is reduced accordingly. However, *how can we ensure the business rules?* There is no strict implementation constraint in the traditional object-oriented method. For example:

From the business perspective, if the approval of the procurement application has been passed, it should be illegal to update purchase items from the procurement application again. However, in the object-oriented scenario, you can't stop programmers from writing code like this:

```
...
PurchaseRequest purchaseRequest = getPurchaseRequest(requestId);
```

```
PurchaseItem item = purchaseRequest.getItem(itemId);
item.setQuantity(1000);
savePurchaseItem(item);
```

Statement 1 made an instance of a purchase requisition, and Statement 2 made an entry in the requisition. Statement 3 and Statement 4 modified purchase requisition entries and saved them. *If purchase requisitions have been approved, wouldn't this modification easily break the budget of the purchase requisitions?*

Certainly, programmers can add logic checks to the code to ensure consistency. Always check the status of `purchaseRequest` before modifying or saving requisition entries, and modification is prohibited if the status is not draft. However, remember the `PurchaseItem` object can be taken out anywhere in the code and may be passed between different methods. If the OO is not properly designed, the business logic may be scattered everywhere. Without design constraints, the implementation of this check is not an easy task.

Let's go back to its essence. *If the purchase item is separated from purchase requisitions, is its own existence valuable?* The answer is no. *If there is no value, does it seem nominal that the modification to the purchase item is essentially a modification to the purchase item, or is it essentially a modification to the purchase requisition?*

If we accept the conclusion that "modifying purchase items is also modifying procurement requisitions," then we should not study purchase items and procurement requisitions separately but should be shown in the following figure:

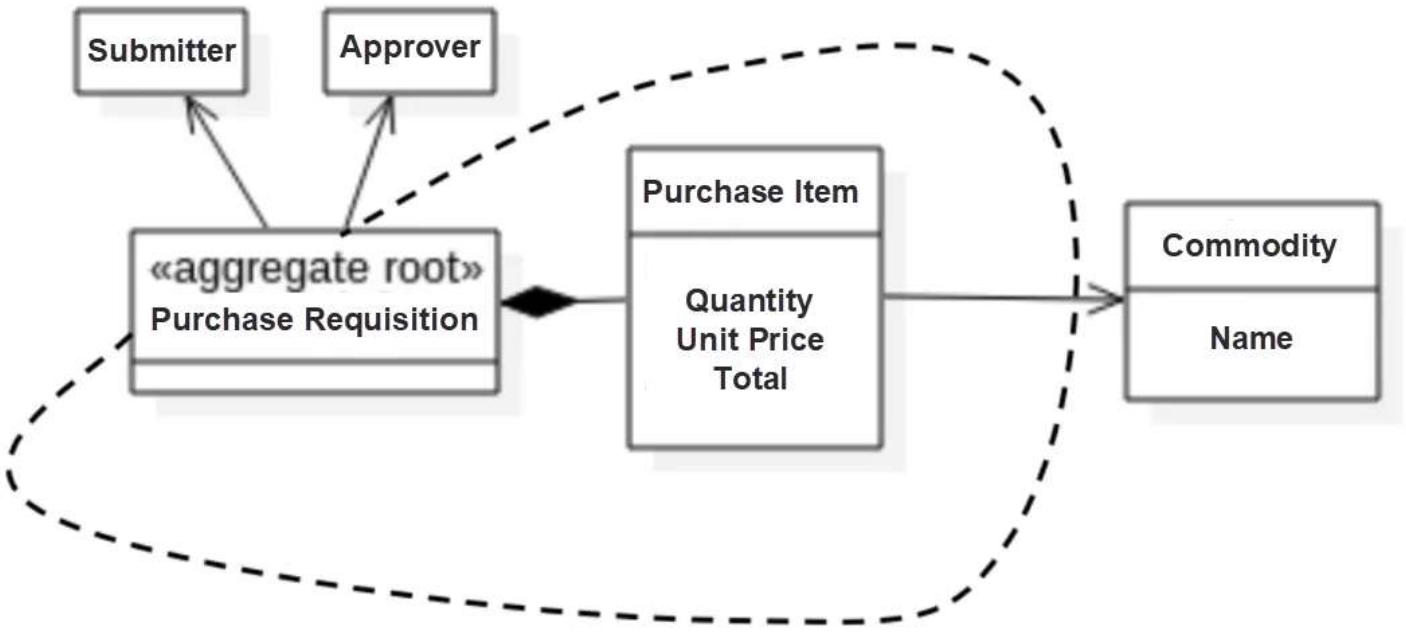


*Figure 3 – Encapsulate Objects with Aggregation*

Organize "purchase requisitions" and "purchase items" together as a larger whole, which is called "aggregation." The business logic of this aggregate should be embedded inside the aggregate. For example, "do not make changes to purchase requisition items after being approved." To achieve this goal, we agree that all operations on purchase items, such as addition, deletion, modification, are operations on purchase requisition objects.

In other words, the `savePurchaseItem()` has never existed in DDD. It should be replaced with `purchaseRequest.modifyPurchaseItem()` and `purchaseRequestRepository.save(purchaseRequest)`.

In the new object relationship, a purchase requisition is responsible for "aggregate root," and purchase items become the aggregated internal data. Since aggregation is now a whole, operations related to it can only be performed through the purchase requisition object, and business consistency can be guaranteed. This is also a more accurate description of the relationship between objects. Although purchase requisitions and purchase items are modeled as objects, their status is unequal. Purchase items are objects subordinate to purchase requisitions, and they are only meaningful if they are a whole.

The essence of aggregation is to establish a boundary greater than the object granularity and gather those closely related objects to form a complete business object. The aggregate root is used as the entry to the external interaction, thus ensuring the consistency of multiple interrelated objects. Reasonable use of aggregation can ensure the consistency of business rules, reduce the possible coupling between objects, improve the intelligibility of design, and reduce the possibility of problems easily.

Therefore, a new layer of encapsulation is constructed above the basic object levels by organizing the objects as aggregation. Encapsulation simplifies concepts and hides details. By doing so, the number of external model elements that need to be concerned about is reduced, thus the complexity is reduced as well. However, the introduction of the encapsulation boundary has also caused some new problems. For example, the product information is also a valid part of purchase items. *Should the product be put into the aggregation of "purchase requisition"? Should the submitter and approver also be put into the aggregation? If we want to obtain the consistency of business rules easily, wouldn't it be better to put all business-related objects together? Are there clear guidelines for putting objects into aggregation and some objects into a non-aggregation?* This article answers these questions in the next section.

## 2. Principles of Aggregation Partition

As one of the layers in the object system of DDD, aggregation also follows the principles of high cohesion and low coupling. This article holds that objects within the aggregation boundary should meet the following heuristic rules:

- Consistency of the lifecycle

- Consistency of the problem domain
- Consistency of the scenario frequency
- As few elements as possible within the aggregation

## 2.1 Consistency of the Lifecycle

Consistency of the lifecycle refers to the objects within the aggregation boundary that have a "personal dependency" relationship with the aggregate root. In other words, if the aggregate root disappears, all other elements in the aggregation should disappear at the same time. For instance, in the preceding example, if the aggregate root (purchase requisition) does not exist, then the purchase item will lose its meaning. The relationship between objects, such as commodities or users as applicants and purchase requisitions, does not exist.

Besides, the consistency of the lifecycle can be proofed by contradiction. If an object remains meaningful after the aggregate root disappears, it means other methods must exist in the system to access the object. This conclusion contradicts the definition of aggregation. Therefore, other elements in the aggregate root must become invalid after the aggregate root disappears. In violation of the consistency of lifecycle, it will also bring serious problems to implementation. Please see the example below:

```
public class PurchaseRequest {
    private Set<PurchaseItem> items;
    private User submitter;
    ...
}
```

In the example, the lifecycle of the user object is inconsistent with the purchase requisition. Let's say there are two segments of code running in parallel:

- **Code 1** obtains the object of a certain purchase requisition, such as a modification to a purchase requisition. Modify this object and save it. *Note:* The User object is also saved because it is embedded in `PurchaseRequest`.

```
r = purchaseRequestRepository.findOne(id);
//...some modifications
purchaseRequestRepository.save(r);
```

- **Code 2** obtains and modifies information about the approver corresponding to this object, such as user management.

```
User user = userRepo.findOne(r.getSubmitter().getId());
//...some modifications
userRepo.save(user);
```

This will lead to a completely unacceptable consequence, which is uncertainty about the modification of User objects! *Therefore, for those objects that are not sure whether they should be included in the same aggregation, do these objects have a separate value if they leave the context of this aggregation?* If the answer is yes, the objects should not be included in this aggregate:

- The User object corresponding to Submitter/Approver is out of the original `PurchaseRequest`, and it still can exist separately.
- The Product object is free from the `PurchaseRequest` and can exist separately.

Therefore, neither of the preceding two objects belongs to the aggregation of the purchase requisition.

## 2.2 Consistency of the Problem Domain

The second principle is the consistency of the problem domain. Problem domain consistency is a constraint of the bounded context. With aggregation as a tactical mode, its models must be within the same bounded context.

Although the first principle shows that the lifecycle consistency of objects can be used as the basis for the partition of aggregation, sometimes it may be controversial whether an object has a meaningful existence apart from another object. For example, *if a purchase requisition is deleted, is the order generated based on this requisition valuable?* Since this example may fall into another argument, it can be avoided from the business process that as long as the order exists, and the purchase requisition cannot be deleted. Let's change this to a very similar example:

On an online forum, users can comment on various articles. These articles should be an aggregate root. If articles are deleted, the users' comments will disappear at the same time. *Can comments belong to articles?*

Now let's consider whether comments may have other uses. For example, a book website allows users to comment on books. If the article aggregation is allowed to hold the comment object simply because there is a logical connection between article deletion and comment deletion, it constrains the scope of application of comments. The clear fact is that comment is essentially far from the article. Therefore, a new principle that overrides principle 1 will be obtained that says objects that do not belong to the same problem domain should not appear in the same aggregation. Readers familiar

with DDD may know that this corresponds to the strategic model of bounded context in DDD. Due to the limited length of the article, this will not be discussed in detail.

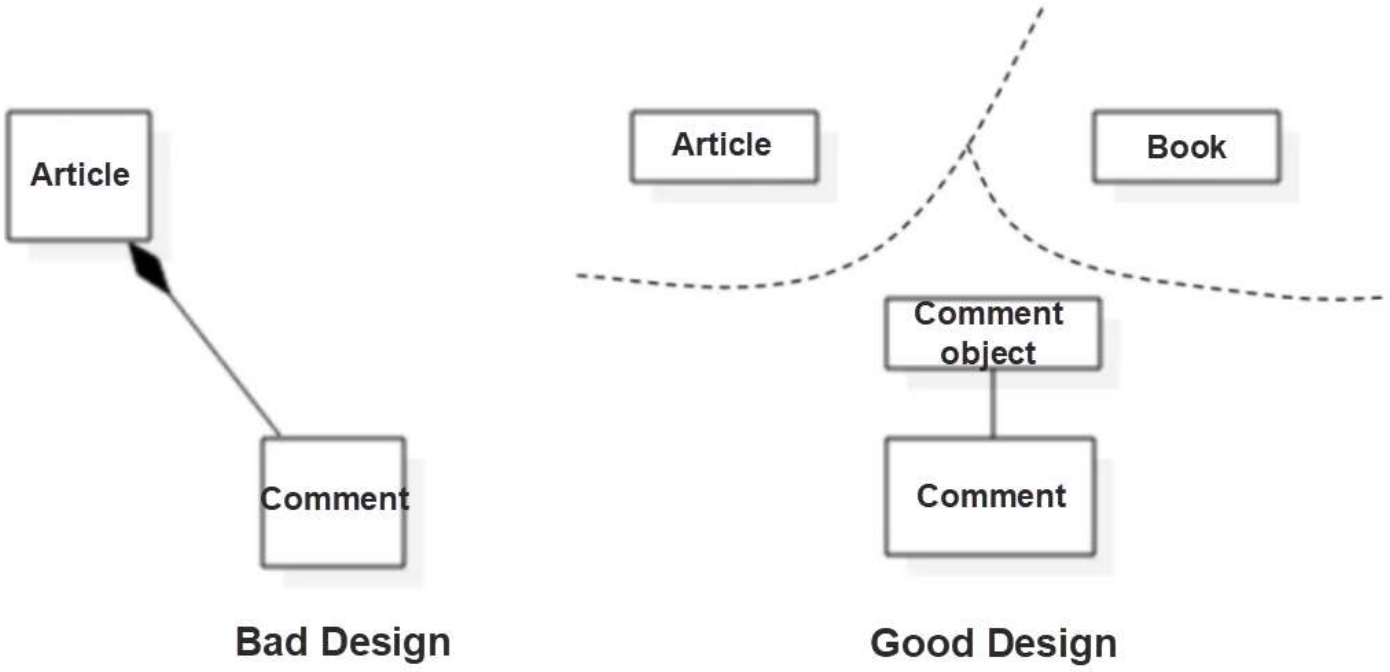


*Figure 4 – Consistency of Problem Domain*

Since an aggregate root cannot guarantee consistency beyond aggregates, we need to rely on the "final consistency" to achieve consistency among aggregates. For example, a message is sent when an article is deleted. The comment system deletes the comment corresponding to the article after receiving the message of deleting the article.

## 2.3 Consistency of the Scenario Frequency

Most aggregations can be distinguished based on the two preceding principles. However, there will still be some complicated situations, for example, the relationship among "product," "version," and "function" in software development. *Are products and versions the same problem domain?* The relationship among these concepts may not be as clear as articles and comments, but it doesn't matter. A heuristic rule exists to avoid this ambiguity, which is called consistency of scenario frequency.

Scenario is a specific description of business, which reflects the way users use the system to achieve business goals. The domain object operations are involved in these scenarios, such as viewing and modifying domain objects. The consistency of scenario frequency is a key characterization of the same internal object of aggregation. Objects that are often operated on at the same time often belong to the same aggregation. Generally, objects that receive very little attention at the same time should not be classified as the same aggregation.

The following figure uses the concepts of product, version, and function as an example. The product contains many functions, and functions are released through a series of versions. However, operations at the product level, such as viewing all the product lists, do not require detailed information about a specific feature or a specific version. When planning the version, the function list is used. However, most of the time, there is no need to check the function details. It is impossible to change the function description when planning the version.

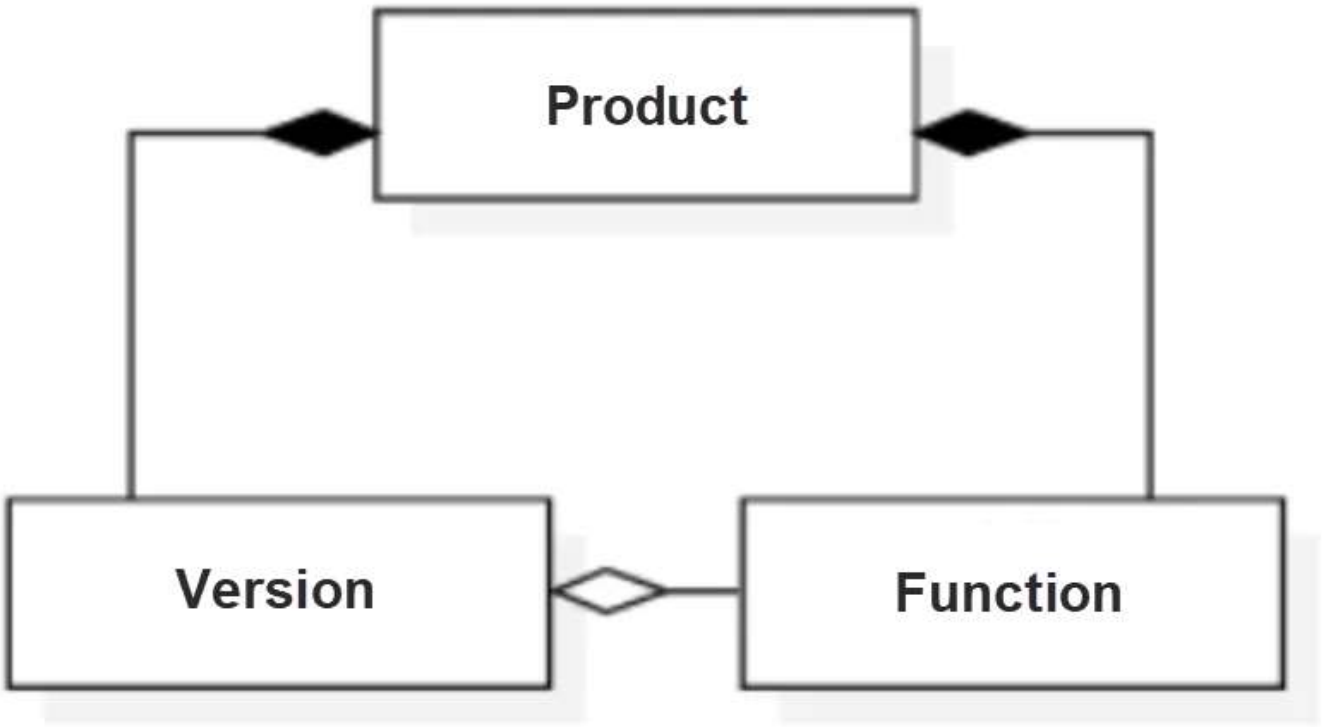


*Figure 5 – Inappropriate Aggregation*

Based on this principle, the following three aggregations have been divided:

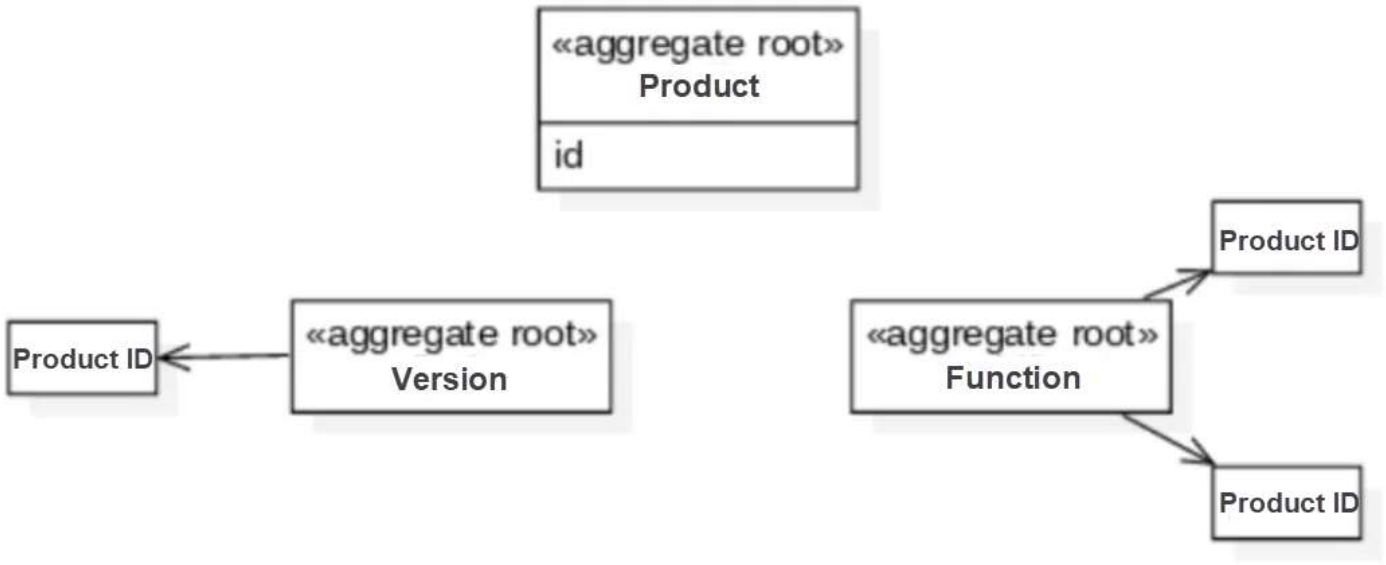


*Figure 6 – More Reasonable Aggregation*

Partitioning aggregations based on scenario consistency is also of great benefit to the implementation. For objects operated in different scenes, putting them into the same aggregation means that each time an object is operated, all the information about the other objects needs to be captured, which is very meaningless. At the implementation level, objects that are not closely related may often be modified concurrently in different scenarios if they appear in the same aggregation, increasing the likelihood of conflicts between these objects. Therefore, objects that operate in inconsistent scenarios (or used in different scenarios) should be considered to divide them into different aggregations.

## 2.4 As Few Elements as Possible within the Aggregation

Aggregation should solve the complexity of consistency. Therefore, it is unnecessary to put all three consistencies in the same aggregation that does not break them. Aggregation only consisting of one business concept, such as the class names, attributes, and the Id objects mentioned shortly in the domain model, is the majority in the object-oriented scenario.

According to the analysis above, in the example of purchase requisition, some attributes of purchase requisition, such as status, submission time, and purchase item, belong to one aggregation. However, products and users cannot be part of this aggregation. *So, how are these aggregations correlated?* A new value object was introduced to solve this problem, as shown in the following figure. The figure also specifies whether the object is a value or an entity.

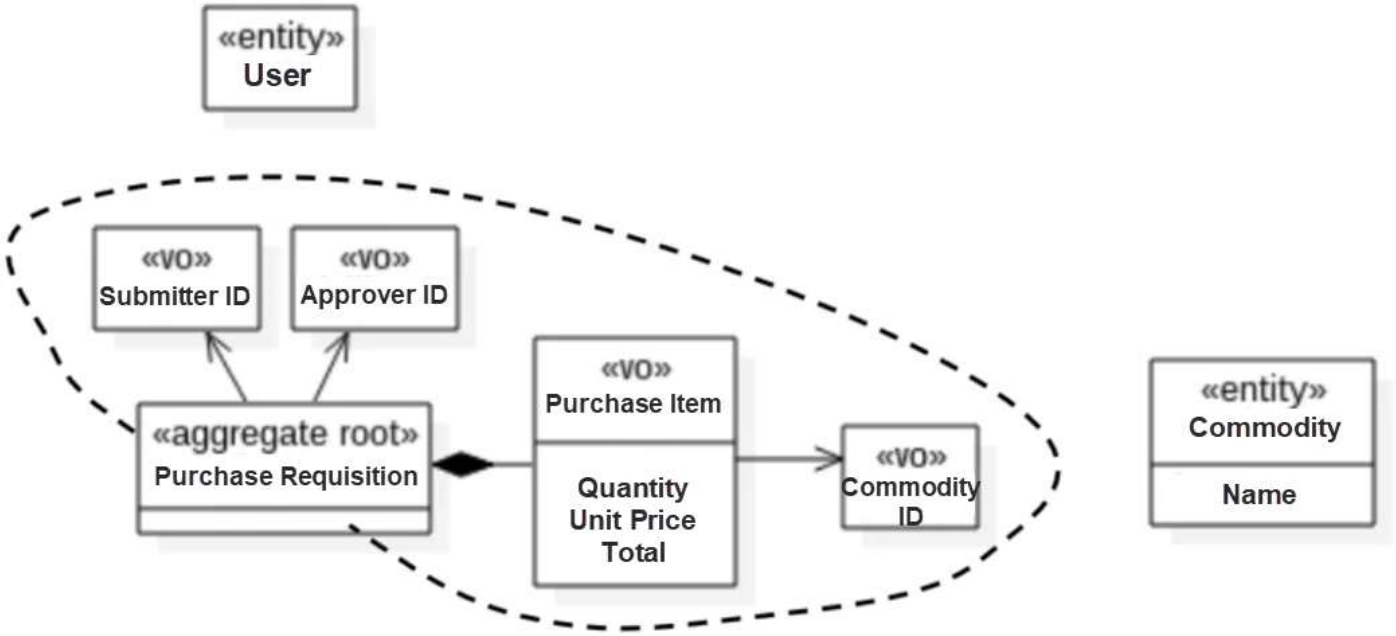


*Figure 7 – Refined Encapsulation Aggregation*

In the aggregation of purchase requisition, besides the aggregate root of a purchase requisition is an entity object, other objects, including Id objects referenced externally. belong to value objects.

The corresponding code is listed below:

```
public class PurchaseRequest {
    private Set<PurchaseItem> items;
    private UserId submitterId;
    ...
}
public class PurchaseItem extends ValueObject{
    private ProductId product;
    private Integer quantity;
    ...
}
```

The introduction of the Id value object is a problem worth discussing.

The introduction of the Id object can stop aggregation and accelerate the speed of the query. However, the information inevitably needs to be queried for the second time in some scenarios. Moreover, the traversal of the EagerFetch/LazyFetch loading mechanism of ORM cannot be used. *Is this a loss?* The answer is no. Do not covet the so-called convenience brought by nested levels of objects that do not belong to an aggregate because it causes far more trouble than benefits. This kind of problem should be solved by external services, such as application layer services.

Besides, does the additional Id value object introduced to break aggregation still count as part of the domain model or "unified language"? My explanation of this problem is that this is part of DDD's implementation mechanism. It belongs to the domain model but controls the visibility in the hands of the development team.

It is unnecessary to communicate these concepts with business personnel but necessary to use the entities, value objects, domain services, and domain events identified by the problem domain to communicate with them. The Id value object, repository, factory, aggregation, and aggregate root are left for the implementation personnel to understand and use in the practices. These concepts are still part of the domain model and the unified language. However, just as views can ignore partial information selectively, these concepts should be ignored in the communication and description with business personnel.

Lastly, please note that this Id object can only reference the Id of other aggregate roots. Since only the aggregate root can be referenced externally, the Id of the aggregate root should be globally unique. The objects inside the aggregation, whether they are entity objects or value objects, only need to ensure that the internal Ids are unique.

# 3. Considerations for Implementation

## 3.1 Aggregation Definitions for Repositories and Factories

The factory and repository modes are specific to DDD, even though in the schema relationship diagram given by the DDD Reference, there are connections between factories, repositories, and entities in addition to aggregations and between factories and value objects. This article reiterates that the strength and value of these connections are different.

The factory mode exists to separate the construction and use of objects, but it contains a deeper meaning in the context of DDD. The direct relationships of the objects in an aggregation can be complex, and service consistency needs to be guaranteed. Then, using a factory to construct aggregated objects is a better encapsulation of complexity. The factory mode is also valuable for the construction of complex entity objects and value objects for non-aggregation. However, this is only a matter of design and implementation and has little to do with the business model.

Although an aggregate factory and a common factory have the same name in factory mode, the factory design based on aggregation is more important for simplifying the complexity of the system. In terms of design constraints, only one factory, which must be visible to the outside, is the aggregation factory. Also, the factory of domain events is meaningful. Domain events are far from the topic of this article and will not be discussed here.

The repository mode does not simply mean data persistence, let alone the database access layer. The more important significance of the repository is that the repository is the storage mechanism of aggregation, and the external world can only complete the access to the aggregation through the repository. It is an aggregated and overall management object. Therefore, an aggregate can have only one repository object in terms of design constraints, which is a repository named after the aggregate root. No other objects should be provided with repository objects.


*Figure 8 – Aggregation and Repository*

## 3.2 The Code Structure Is Consistent with the Aggregation

Careful readers have found that the package organization in the figure above is also consistent with the aggregation, and the name of the aggregation root is used as the package name. This is my custom way of organizing code. I take aggregation as one level of code and put all entity objects, including aggregate roots, value objects, repositories, and factories that belong to the aggregation in the same code package. *Remember, there are other layers above this, such as bounding contexts and modules.* The code structure is highly consistent with the structure of the domain model, reducing the representation gap and managing the complexity of the object scenario.

## 3.3 Aggregation Cannot Cross the Deployment Boundary

Deployment boundary is a complex topic. This article only discusses content related to aggregation. First, if the system adopts microservices architecture, the deployment boundary and the bounded context boundary should be kept consistent, instead of making the granularity of the deployment larger than that of the bounded context, which can bring better business flexibility and scalability.

From the minimum boundary of the service, the minimum boundary must not be smaller than the granularity of aggregation. Otherwise, it will bring a large number of data consistency problems because the consistency between microservices needs to be ensured through eventual consistency. If aggregation crosses the deployment boundary, it will be a consistency disaster. I have seen some unreasonable suggestions on the partition of microservices in some books, such as making the addition, deletion, modification, and query of each object a service. This kind of suggestion is wrong in my opinion.

## 3.4 Aggregation Improves System Performance and Scalability

Many people are troubled by inefficient queries in the ORM mechanism. The reason for this is clear when considering the previous examples. Let's add Spring JPA annotations to the preceding incorrect aggregation example:

```
public class PurchaseRequest {
    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    private Set<PurchaseItem> items;
    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    private User submitter;
    ...
}
```

Due to a lack of the concept of aggregation or an incorrect super-large aggregation, each query to `PurchaseRequest` needs to capture a large number of objects from the system, consuming a lot of computing resources. *Is the User is also a very large object?* However, it also has its own problems, so the performance cannot be good.

Maybe some readers will consider using Lazy Fetch rather than Eager Fetch. Indeed, this is better for performance, but unfortunately, the context of data access will have to be retained all the time. For this reason, the probability of system failures increases significantly, bringing inconvenience to the distributed design.

Small aggregation does not have this problem at all. Under this condition, each object involved in access, the aggregation actually, cannot be very large, and the required data is appropriately there. Therefore, data and service integrity are guaranteed, and horizontal scaling is easily performed to ensure performance and scalability.

# 4. Summary

Modeling is one of the ways we understand the real world and simplify the complexity of problems. Aggregation, as a level in domain modeling, enables information hiding, increases the abstraction level, and encapsulates closely related business logic through appropriate boundaries. As a result, the consistency of system data is ensured, and the system performance is improved.

This article discusses the definition and value of aggregation:

- Aggregation is a hierarchy of modeling in the object-oriented scenario. It hides fine-grained objects and constrains the coupling between objects.
- Aggregation is the boundary of consistency and the encapsulation of closely related objects. Aggregation encapsulates entity objects and value objects and takes the most important entity object as the aggregate root. As the only external portal of aggregation, the aggregate root ensures the consistency of business rules and data.

This article also discusses four heuristic rules for aggregation recognition:

- Consistency of the lifecycle
- Consistency of the problem domain
- Consistency of the scenario frequency
- As few elements as possible within the aggregation

From the perspective of implementation, the granularity of repository and factory should be the same as aggregation, and the structure of code and deployment can also be aligned with aggregation. Besides, implementation is consistent with the domain model, which is also the goal and value of domain-driven design as a correct OO.