Effective Aggregate Design
Part I: Modeling a Single Aggregate

Vaughn Vernon: vvernon@shiftmethod.com

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

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

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

Designing a Scrum Management Application

The best way to explain **aggregates** is with an example. Our fictitious company is developing an application to support Scrum-based projects, *ProjectOvation*. It follows the traditional Scrum project management model, complete with product, product owner, team, backlog items, planned releases, and sprints. If you think of Scrum at its richest, that's where *ProjectOvation* is headed. This provides a familiar domain to most of us. The Scrum terminology forms the starting point of the **ubiquitous language**. It is a subscription-based application hosted using the software as a service (SaaS) model. Each subscribing organization is registered as a *tenant*, another term for our **ubiquitous language**.

The company has assembled a group of talented Scrum experts and Java developers.[1] However, their experience with DDD is somewhat limited. That means the team is going to make some mistakes with DDD as they climb a difficult learning curve. They will grow, and so can we. Their struggles may help us recognize and change similar unfavorable situations we've created in our own software.

1 Although the examples use Java and Hibernate, all of this material is applicable to C# and NHibernate, for instance.

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

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

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

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

First Attempt: Large-Cluster Aggregate

The team put a lot of weight on the words "Products have" in the first statement. It sounded to some like composition, that objects needed to be interconnected like an object graph. Maintaining these object life cycles together was considered very important. So, the developers added the following consistency rules into the specification:

- If a backlog item is committed to a sprint, we must not allow it to be removed from the system.
- If a sprint has committed backlog items, we must not allow it to be removed from the system.
- If a release has scheduled backlog items, we must not allow it to be removed from the system.
- If a backlog item is scheduled for release, we must not allow it to be removed from the system.

As a result, `Product` was first modeled as a very large **aggregate**. The **root** object, `Product`, held all `BacklogItem`, all `Release`, and all `Sprint` instances associated with it. The interface design protected all parts from inadvertent client removal. This design is shown in the following code, and as a UML diagram in Figure 1:

```
public class Product extends ConcurrencySafeEntity  {
    private Set<BacklogItem> backlogItems;
    private String description;
    private String name;
    private ProductId productId;
    private Set<Release> releases;
    private Set<Sprint> sprints;
    private TenantId tenantId;
    ...
}
```

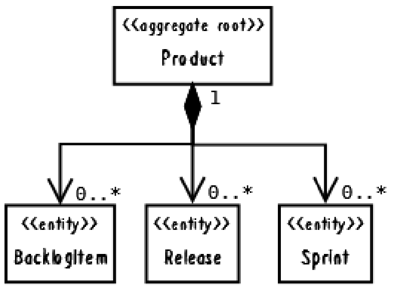

Figure 1: `Product` modeled as a very large **aggregate**.

The big **aggregate** looked attractive, but it wasn't truly practical. Once the application was running in its intended multi-user environment it began to regularly experience transactional failures. Let's look more closely at a few client usage patterns and how they interact with our technical solution model. Our **aggregate** instances employ optimistic concurrency to protect persistent objects from simultaneous overlapping modifications by different clients, thus avoiding the use of database locks. Objects carry a version number that is incremented when changes are made and checked before they are saved to the database. If the version on the persisted object is greater than the version on the client's copy, the client's is considered stale and updates are rejected.

Consider a common simultaneous, multi-client usage scenario:

- Two users, Bill and Joe, view the same `Product` marked as version 1, and begin to work on it.
- Bill plans a new `BacklogItem` and commits. The `Product` version is incremented to 2.
- Joe schedules a new `Release` and tries to save, but his commit fails because it was based on `Product` version 1.

Persistence mechanisms are used in this general way to deal with concurrency.[2] If you argue that the default concurrency configurations can be changed, reserve your verdict for a while longer. This approach is actually important to protecting **aggregate** invariants from concurrent changes.

These consistency problems came up with just two users. Add more users, and this becomes a really big problem. With Scrum, multiple users often make these kinds of overlapping modifications during the sprint planning meeting and in sprint execution. Failing all but one of their requests on an ongoing basis is completely unacceptable.

Nothing about planning a new backlog item should logically interfere with scheduling a new release! Why did Joe's commit fail? At the heart of the issue, the large cluster **aggregate** was designed with false invariants in mind, not real business rules. These false invariants are artificial constraints imposed by developers. There are other ways for the team to prevent inappropriate removal without being arbitrarily restrictive. Besides causing transactional issues, the design also has performance and scalability drawbacks.

Second Attempt: Multiple Aggregates

Now consider an alternative model as shown in Figure 2, in which we have four distinct **aggregates**. Each of the dependencies is associated by inference using a common `ProductId`, which is the identity of `Product` considered the parent of the other three.

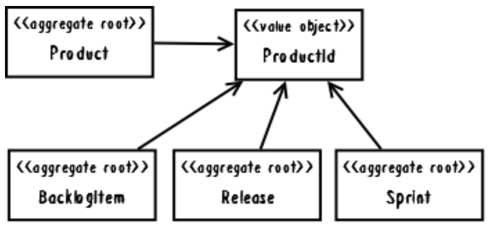

Figure 2: `Product` and related concepts are modeled as separate **aggregate** types.

Breaking the large **aggregate** into four will change some method contracts on `Product`. With the large cluster **aggregate** design the method signatures looked like this:

```
public class Product ... {
    ...
    public void planBacklogItem(
        String aSummary, String aCategory,
        BacklogItemType aType, StoryPoints aStoryPoints) {
            ...
    }
    ...
```

2 For example, Hibernate provides optimistic concurrency in this way. The same could be true of a key-value store because the entire **aggregate** is often serialized as one value, unless designed to save composed parts separately.

```
    public void scheduleRelease(
        String aName, String aDescription,
        Date aBegins, Date anEnds) {
        ...
    }

    public void scheduleSprint(
        String aName, String aGoals,
        Date aBegins, Date anEnds) {
        ...
    }
    ...
}
```

All of these methods are [CQS] commands. That is, they modify the state of the `Product` by adding the new element to a collection, so they have a `void` return type. But with the multiple **aggregate** design, we have:

```
public class Product ... {
    ...
    public BacklogItem planBacklogItem(
        String aSummary, String aCategory,
        BacklogItemType aType, StoryPoints aStoryPoints) {
        ...
    }

    public Release scheduleRelease(
        String aName, String aDescription,
        Date aBegins, Date anEnds) {
        ...
    }

    public Sprint scheduleSprint(
        String aName, String aGoals,
        Date aBegins, Date anEnds) {
        ...
    }
    ...
}
```

These redesigned methods have a [CQS] query contract, and act as **factories**. That is, they each respectively create a new **aggregate** instance and return a reference to it. Now when a client wants to plan a backlog item, the transactional **application service** must do the following:

```
public class ProductBacklogItemService ... {
    ...
    @Transactional
    public void planProductBacklogItem(
        String aTenantId, String aProductId,
        String aSummary, String aCategory,
        String aBacklogItemType, String aStoryPoints) {

        Product product =
            productRepository.productOfId(
                    new TenantId(aTenantId),
                    new ProductId(aProductId));

        BacklogItem plannedBacklogItem =
            product.planBacklogItem(
                    aSummary,
                    aCategory,
                    BacklogItemType.valueOf(aBacklogItemType),
                    StoryPoints.valueOf(aStoryPoints));

        backlogItemRepository.add(plannedBacklogItem);
    }
    ...
}
```

So we've solved the transaction failure issue *by modeling it away*. Any number of `BacklogItem`, `Release`, and `Sprint` instances can now be safely created by simultaneous user requests. That's pretty simple.

However, even with clear transactional advantages, the four smaller **aggregates** are less convenient from the perspective of client consumption. Perhaps instead we could tune the large **aggregate** to eliminate the concurrency issues. By setting our Hibernate mapping `optimistic-lock` option to `false`, the transaction failure domino effect goes away. There is no invariant on the total number of created `BacklogItem`, `Release`, or `Sprint` instances, so why not just allow the collections to grow unbounded and ignore these specific modifications on `Product`? What additional cost would there be in keeping the large cluster **aggregate**? The problem is that it could actually grow out of control. Before thoroughly examining why, let's consider the most important modeling tip the team needed.

Rule: Model True Invariants In Consistency Boundaries

When trying to discover the **aggregates** in a **bounded context**, we must understand the model's true invariants. Only with that knowledge can we determine which objects should be clustered into a given **aggregate**.

An invariant is a business rule that must always be consistent. There are different kinds of consistency. One is transactional, which is considered immediate and atomic. There is also eventual consistency. When discussing invariants, we are referring to transactional consistency. We might have the invariant:

```
c = a + b
```

Therefore, when `a` is 2 and `b` is 3, `c` must be 5. According to that rule and conditions, if `c` is anything but 5, a system invariant is violated. To ensure that `c` is consistent, we model a boundary around these specific attributes of the model:

```
AggregateType1 {
  int a; int b; int c;
  operations...
}
```

The consistency boundary logically asserts that everything inside adheres to a specific set of business invariant rules no matter what operations are performed. The consistency of everything outside this boundary is irrelevant to the **aggregate**. Thus, **aggregate** is synonymous with transactional consistency boundary. (In this limited example, `AggregateType1` has three attributes of type `int`, but any given **aggregate** could hold attributes of various types.)

When employing a typical persistence mechanism we use a single transaction[3] to manage consistency. When the transaction commits, everything inside one boundary must be consistent. A properly designed **aggregate** is one that can be modified in any way required by the business with its invariants completely consistent within a single transaction. And a properly designed **bounded context** modifies only one **aggregate** instance per transaction in all cases. What is more, we cannot correctly reason on **aggregate** design without applying transactional analysis.

Limiting the modification of one **aggregate** instance per transaction may sound overly strict. However, it is a rule of thumb and should be the goal in most cases. It addresses the very reason to use **aggregates**.

Since **aggregates** must be designed with a consistency focus, it implies that the user interface should concentrate each request to execute a single command on just one **aggregate** instance. If user requests try to accomplish too much, it will force the application to modify multiple instances at once.

Therefore, **aggregates** are chiefly about consistency boundaries and not driven by a desire to design object graphs. Some real-world invariants will be more complex than this. Even so, typically invariants will be less demanding on our modeling efforts, making it possible to *design small **aggregates***.

Rule: Design Small Aggregates

We can now thoroughly address the question: What additional cost would there be in keeping the large cluster **aggregate**? Even if we guarantee that every transaction would succeed, we still limit performance and scalability. As our company develops its market, it's going to bring in lots of tenants. As each tenant makes a deep commitment to *ProjectOvation*, they'll host more and more projects and the management artifacts to go along with them. That will result in vast numbers of products, backlog items, releases, sprints, and others. Performance and scalability are non-functional requirements that cannot be ignored.

Keeping performance and scalability in mind, what happens when one user of one tenant wants to add a single backlog item to a product, one that is years old and already has thousands of backlog items? Assume a persistence mechanism capable of lazy loading (Hibernate). We almost never load all backlog items, releases, and sprints all at once. Still, thousands of backlog items would be loaded into memory just to add one new element to the already large collection. It's worse if a persistence mechanism does not support lazy loading. Even being memory conscious, sometimes we would have to load multiple collections, such as when scheduling a backlog item for release or committing one to a sprint; all backlog items, and either all releases or all sprints, would be loaded.

3 The transaction may be handled by a **unit of work**.

To see this clearly, look at the diagram in Figure 3 containing the zoomed composition. Don't let the 0..* fool you; the number of associations will almost never be zero and will keep growing over time. We would likely need to load thousands and thousands of objects into memory all at once, just to carry out what should be a relatively basic operation. That's just for a single team member of a single tenant on a single product. We have to keep in mind that this could happen all at once with hundreds and thousands of tenants, each with multiple teams and many products. And over time the situation will only become worse.

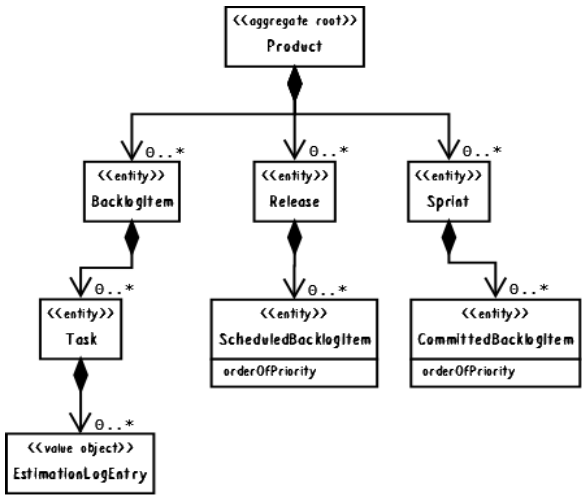

Figure 3: With this `Product` model, multiple large collections load during many basic operations.

This large cluster **aggregate** will never perform or scale well. It is more likely to become a nightmare leading only to failure. It was deficient from the start because the false invariants and a desire for compositional convenience drove the design, to the detriment of transactional success, performance, and scalability.

If we are going to design small **aggregates**, what does "small" mean? The extreme would be an **aggregate** with only its globally unique identity and one additional attribute, which is not what's being recommended (unless that is truly what one specific **aggregate** requires). Rather, limit the **aggregate** to just the **root entity** and a minimal number of attributes and/or **value**-typed properties.[4] The correct minimum is the ones necessary, and no more.

Which ones are necessary? The simple answer is: those that must be consistent with others, even if domain experts don't specify them as rules. For example, `Product` has `name`

4 A value-typed property is an attribute that holds a reference to a **value object**. I distinguish this from a simple attribute such as a String or numeric type, as does Ward Cunningham when describing Whole Value; see http://fit.c2.com/wiki.cgi?WholeValue

and `description` attributes. We can't imagine `name` and `description` being inconsistent, modeled in separate **aggregates**. When you change the `name` you probably also change the `description`. If you change one and not the other, it's probably because you are fixing a spelling error or making the `description` more fitting to the `name`. Even though domain experts will probably not think of this as an explicit business rule, it is an implicit one.

What if you think you should model a contained part as an **entity**? First ask whether that part must itself change over time, or whether it can be completely replaced when change is necessary. If instances can be completely replaced, it points to the use of a **value object** rather than an **entity**. At times **entity** parts are necessary. Yet, if we run through this exercise on a case-by-case basis, many concepts modeled as **entities** can be refactored to **value objects**. Favoring **value** types as **aggregate** parts doesn't mean the **aggregate** is immutable since the **root entity** itself mutates when one of its **value**-typed properties is replaced.

There are important advantages to limiting internal parts to **values**. Depending on your persistence mechanism, **values** can be serialized with the **root entity**, whereas **entities** usually require separately tracked storage. Overhead is higher with **entity** parts, as, for example, when SQL joins are necessary to read them using Hibernate. Reading a single database table row is much faster. **Value objects** are smaller and safer to use (fewer bugs). Due to immutability it is easier for unit tests to prove their correctness.

On one project for the financial derivatives sector using [Qi4j], Niclas [Hedhman] reported that his team was able to design approximately 70% of all **aggregates** with just a **root entity** containing some **value**-typed properties. The remaining 30% had just two to three total **entities**. This doesn't indicate that all domain models will have a 70/30 split. It does indicate that a high percentage of **aggregates** can be limited to a single **entity**, the **root**.

The [DDD] discussion of **aggregates** gives an example where multiple **entities** makes sense. A purchase order is assigned a maximum allowable total, and the sum of all line items must not surpass the total. The rule becomes tricky to enforce when multiple users simultaneously add line items. Any one addition is not permitted to exceed the limit, but concurrent additions by multiple users could collectively do so. I won't repeat the solution here, but I want to emphasize that most of the time the invariants of business models are simpler to manage than that example. Recognizing this helps us to model **aggregates** with as few properties as possible.

Smaller **aggregates** not only perform and scale better, they are also biased toward transactional success, meaning that conflicts preventing a commit are rare. This makes a system more usable. Your domain will not often have true invariant constraints that force you into large composition design situations. Therefore, it is just plain smart to limit **aggregate** size. When you occasionally encounter a true consistency rule, then add another few **entities**, or possibly a collection, as necessary, but continue to push yourself to keep the overall size as small as possible.

Don't Trust Every Use Case

Business analysts play an important role in delivering use case specifications. Since much work goes into a large and detailed specification, it will affect many of our design decisions. Yet, we mustn't forget that use cases derived in this way does not carry the perspective of the domain experts and developers of our close-knit modeling team. We still must reconcile each use case with our current model and design, including our decisions about **aggregates**. A common issue that arises is a particular use case that calls for the modification of multiple **aggregate** instances. In such a case we must determine whether the specified large user goal is spread across multiple persistence transactions, or if it occurs within just one. If it is the latter, it pays to be skeptical. No matter how well it is written, such a use case may not accurately reflect the true **aggregates** of our model.

Assuming your **aggregate** boundaries are aligned with real business constraints, then it's going to cause problems if business analysts specify what you see in Figure 4. Thinking through the various commit order permutations, you'll see that there are cases where two of the three requests will fail.[5] What does attempting this indicate about your design? The answer to that question may lead to a deeper understanding of the domain. Trying to keep multiple **aggregate** instances consistent may be telling you that your team has missed an invariant. You may end up folding the multiple **aggregates** into one new concept with a new name in order to address the newly recognized business rule. (And, of course, it might be only parts of the old **aggregates** that get rolled into the new one.)

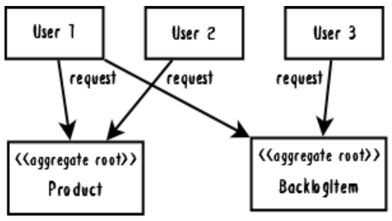

Figure 4: Concurrency contention exists between three users all trying to access the same two **aggregate** instances, leading to a high number of transactional failures.

5 This doesn't address the fact that some use cases describe modifications to multiple **aggregates** that span transactions, which would be fine. A user goal should not be viewed as synonymous with transaction. We are only concerned with use cases that actually indicate the modification of multiple **aggregates** instances in one transaction.

So a new use case may lead to insights that push us to remodel the **aggregate**, but be skeptical here, too. Forming one **aggregate** from multiple ones may drive out a completely new concept with a new name, yet if modeling this new concept leads you toward designing a large cluster **aggregate**, that can end up with all the problems of large **aggregates**. What different approach may help?

Just because you are given a use case that calls for maintaining consistency in a single transaction doesn't mean you should do that. Often, in such cases, the business goal can be achieved with *eventual consistency* between **aggregates**. The team should critically examine the use cases and challenge their assumptions, especially when following them as written would lead to unwieldy designs. The team may have to rewrite the use case (or at least re-imagine it if they face an uncooperative business analyst). The new use case would specify *eventual consistency and the acceptable update delay*. This is one of the issues taken up in Part II of this essay.

Coming in Part II

Part I has focused on the design of a number of small **aggregates** and their internals. There will be cases that require references and consistency between **aggregates**, especially when we keep **aggregates** small. Part II of this essay covers how **aggregates** reference other **aggregates** as well as eventual consistency.

Acknowledgments

Eric Evans and Paul Rayner did several detailed reviews of this essay. I also received feedback from Udi Dahan, Greg Young, Jimmy Nilsson, Niclas Hedhman, and Rickard Öberg.

References

[CQS] Martin Fowler explains Bertrand Meyer's Command-Query Separation: http://martinfowler.com/bliki/CommandQuerySeparation.html

[DDD] Eric Evans; *Domain-Driven Design—Tackling Complexity in the Heart of Software;* 2003, Addison-Wesley, ISBN 0-321-12521-5.

[Hedhman] Niclas Hedhman; http://www.jroller.com/niclas/

[Qi4j] Rickard Öberg, Niclas Hedhman; Qi4j framework; http://qi4j.org/

Biography

Vaughn Vernon is a veteran consultant, providing architecture, development, mentoring, and training services. This three-part essay is based on his upcoming book on implementing domain-driven design. His *QCon San Francisco 2010* presentation on **context mapping** is available on the DDD Community site: http://dddcommunity.org/library/vernon_2010. Vaughn blogs here: http://vaughnvernon.co/, and you can reach him by email here: vvernon@shiftmethod.com