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.

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.

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

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;
    ...
}
```

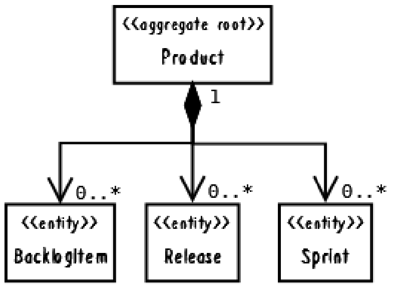

Figure 1: `Product` modeled as a very large **aggregate**.

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.

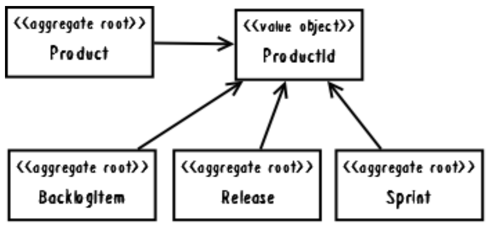

Figure 2: `Product` and related concepts are modeled as separate **aggregate** types.

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.

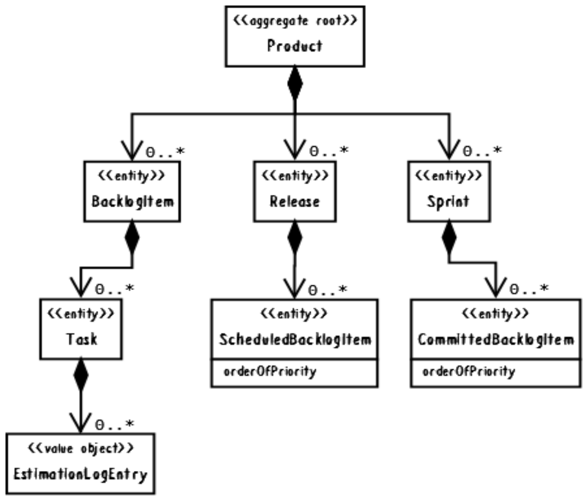

Figure 3: With this `Product` model, multiple large collections load during many basic operations.

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.)

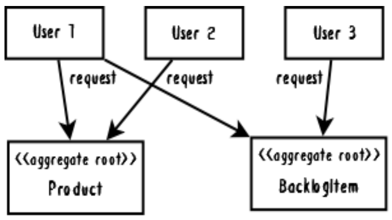

Figure 4: Concurrency contention exists between three users all trying to access the same two **aggregate** instances, leading to a high number of transactional failures.

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

Effective Aggregate Design
Part II: Making Aggregates Work Together

Vaughn Vernon: vvernon@shiftmethod.com

Part I focused on the design of a number of small **aggregates** and their internals. In Part II we discuss how **aggregates** reference other **aggregates**, as well as how to leverage eventual consistency to keep separate **aggregate** instances in harmony.

When designing **aggregates**, we may desire a compositional structure that allows for traversal through deep object graphs, but that is not the motivation of the pattern. [DDD] states that one **aggregate** may hold references to the **root** of other **aggregates**. However, we must keep in mind that this does not place the referenced **aggregate** inside the consistency boundary of the one referencing it. The reference does not cause the formation of just one, whole **aggregate**. There are still two (or more), as shown in Figure 5.

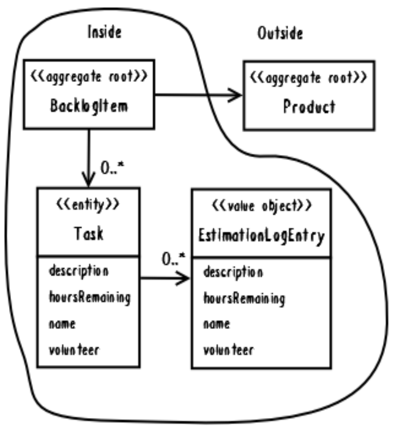

Figure 5: There are two **aggregates**, not one.

In Java the association would be modeled like this:

```
public class BacklogItem extends ConcurrencySafeEntity  {
    ...
    private Product product;
    ...
}
```

That is, the `BacklogItem` holds a direct object association to `Product`.

In combination with what's already been discussed and what's next, this has a few implications:

1. Both the referencing **aggregate** (`BacklogItem`) and the referenced **aggregate** (`Product`) *must not* be modified in the same transaction. Only one or the other may be modified in a single transaction.
2. If you are modifying multiple instances in a single transaction, it may be a strong indication that your consistency boundaries are wrong. If so, it is possibly a missed modeling opportunity; a concept of your **ubiquitous language** has not yet been discovered although it is waving its hands and shouting at you (see Part I).
3. If you are attempting to apply point #2, and doing so influences a large cluster **aggregate** with all the previously stated caveats, it may be an indication that you need to use *eventual consistency* (see below) instead of atomic consistency.

If you don't hold any reference, you can't modify another **aggregate**. So the temptation to modify multiple **aggregates** in the same transaction could be squelched by avoiding the situation in the first place. But that is overly limiting since domain models always require some associative connections. What might we do to facilitate necessary associations, protect from transaction misuse or inordinate failure, and allow the model to perform and scale?

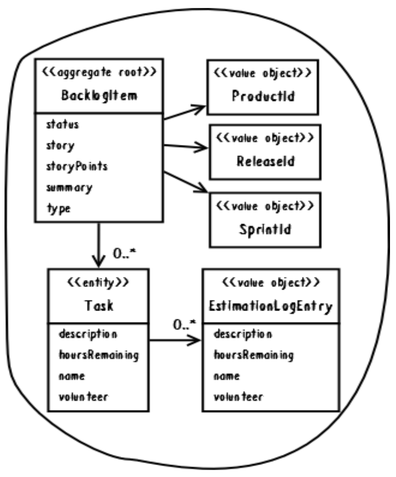

Figure 6: The `BacklogItem` **aggregate**, inferring associations outside its boundary with identities.

Rule: Reference Other Aggregates By Identity

Prefer references to external **aggregates** only by their globally unique identity, not by holding a direct object reference (or “pointer”). This is exemplified in Figure 6. We would refactor the source to:

```
public class BacklogItem extends ConcurrencySafeEntity  {
    ...
    private ProductId productId;
    ...
}
```

Aggregates with inferred object references are thus automatically smaller because references are never eagerly loaded. The model can perform better because instances require less time to load and take less memory. Using less memory has positive implications both for memory allocation overhead and garbage collection.

Model Navigation

Reference by identity doesn't completely prevent navigation through the model. Some will use a **repository** from inside an **aggregate** for look up. This technique is called **disconnected domain model**, and it's actually a form of lazy loading. There's a different recommended approach, however: Use a **repository** or **domain service** to look up dependent objects ahead of invoking the **aggregate** behavior. A client **application service** may control this, then dispatch to the **aggregate**:

```
public class ProductBacklogItemService ... {
    ...
    @Transactional
    public void assignTeamMemberToTask(
        String aTenantId,
        String aBacklogItemId,
        String aTaskId,
        String aTeamMemberId) {

        BacklogItem backlogItem =
            backlogItemRepository.backlogItemOfId(
                new TenantId(aTenantId),
                new BacklogItemId(aBacklogItemId));

        Team ofTeam =
            teamRepository.teamOfId(
                backlogItem.tenantId(),
                backlogItem.teamId());

        backlogItem.assignTeamMemberToTask(
                new TeamMemberId(aTeamMemberId),
                ofTeam,
                new TaskId(aTaskId));
    }
    ...
}
```

Having an **application service** resolve dependencies frees the **aggregate** from relying on either a **repository** or a **domain service**. Again, referencing multiple **aggregates** in one request does not give license to cause modification on two or more of them.

Limiting a model to using reference only by identity could make it more difficult to serve clients that assemble and render **user interface** views. You may have to use multiple **repositories** in a single use case to populate views. If query overhead causes performance issues, it may be worth considering the use of **CQRS** [Dahan, Fowler, Young]. Or you may need to strike a balance between inferred and direct object reference.

If all this advice seems to lead to a less convenient model, consider the additional benefits it affords. Making **aggregates** smaller leads to better performing models, plus we can add scalability and distribution.

Scalability and Distribution

Since **aggregates** don't use direct references to other **aggregates**, but reference by identity, their persistent state can be moved around to reach large scale. *Almost-infinite scalability* is achieved by allowing for continuous repartitioning of **aggregate** data storage, as explained by Amazon.com's Pat [Helland] in his position paper, *Life Beyond Distributed Transactions: an Apostate's Opinion*. What we call **aggregate** he calls *entity*. But what he describes is still **aggregate** by any other name; a unit of composition that has transactional consistency. Some NoSql persistence mechanisms support the Amazon-inspired distributed storage. These provide much of what [Helland] refers to as the lower, scale-aware layer. When employing a distributed store, or even when using a SQL database with similar motivations, reference by identity plays an important role.

Distribution extends beyond storage. Since there are always multiple **bounded contexts** at play in a given **core domain** initiative, reference by identity allows distributed domain models to have associations from afar. When an event-driven approach is in use, message-based **domain events** containing **aggregate** identities are sent around the enterprise. Message subscribers in foreign **bounded contexts** use the identities to carry out operations in their own domain models. Reference by identity forms remote associations or *partners*. Distributed operations are managed by what [Helland] calls *two-party activities*; but in **publish-subscribe** [POSA1, GoF] terms it's *multi-party* (two or more). Transactions across distributed systems are not atomic. The various systems bring multiple **aggregates** into a consistent state eventually.

Rule: Use Eventual Consistency Outside the Boundary

There is a frequently overlooked statement found in the [DDD] **aggregate** pattern definition. It bears heavily on what we must do to achieve model consistency when multiple **aggregates** must be affected by a single client request.

DDD p128: Any rule that spans AGGREGATES will not be expected to be up-to-date at all times. Through event processing, batch processing, or other update mechanisms, other dependencies can be resolved within some specific time.

Thus, if executing a command on one **aggregate** instance requires that additional business rules execute on one or more other **aggregates**, use *eventual consistency*. Accepting that all **aggregate** instances in a large-scale, high-traffic enterprise are never completely consistent helps us accept that eventual consistency also makes sense in the smaller scale where just a few instances are involved.

Ask the domain experts if they could tolerate some time delay between the modification of one instance and the others involved. Domain experts are sometimes far more comfortable with the idea of delayed consistency than are developers. They are aware of realistic delays that occur all the time in their business, whereas developers are usually indoctrinated with an atomic change mentality. Domain experts often remember the days prior to computer automation of their business operations, when various kinds of delays occurred all the time and consistency was never immediate. Thus, domain experts are often willing to allow for reasonable delays—a generous number of seconds, minutes, hours, or even days—before consistency occurs.

There is a practical way to support eventual consistency in a DDD model. An **aggregate** command method publishes a **domain event** that is in time delivered to one or more asynchronous subscribers:

```
public class BacklogItem extends ConcurrencySafeEntity  {
    ...
    public void commitTo(Sprint aSprint) {
        ...
        DomainEventPublisher
            .instance()
            .publish(new BacklogItemCommitted(
                    this.tenantId(),
                    this.backlogItemId(),
                    this.sprintId()));
    }
    ...
}
```

These subscribers each then retrieve a different yet corresponding **aggregate** instance and execute their behavior based on it. Each of the subscribers executes in a separate transaction, obeying the rule of **aggregate** to modify just one instance per transaction.

What happens if the subscriber experiences concurrency contention with another client, causing its modification to fail? The modification can be retried if the subscriber does not acknowledge success to the messaging mechanism. The message will be redelivered, a new transaction started, a new attempt made to execute the necessary command, and a corresponding commit. This retry process can continue until consistency is achieved, or until a retry limit is reached. If complete failure occurs it may be necessary to compensate, or at a minimum to report the failure for pending intervention.

What is accomplished by publishing the `BacklogItemCommitted` **event** in this specific example? Recalling that `BacklogItem` already holds the identity of the `Sprint` it is committed to, we are in no way interested in maintaining a meaningless bidirectional association. Rather, the **event** allows for the eventual creation of a `CommittedBacklogItem` so the `Sprint` can make a record of work commitment. Since each `CommittedBacklogItem` has an `ordering` attribute, it allows the `Sprint` to give each `BacklogItem` an ordering different than `Product` and `Release` have, and that is not tied to the `BacklogItem` instance's own recorded estimation of `BusinessPriority`. Thus, `Product` and `Release` each hold similar associations, namely `ProductBacklogItem` and `ScheduledBacklogItem`, respectively.

This example demonstrates how to use eventual consistency in a single **bounded context**, but the same technique can also be applied in a distributed fashion as previously described.

Ask Whose Job It Is

Some domain scenarios can make it very challenging to determine whether transactional or eventual consistency should be used. Those who use DDD in a classic/traditional way may lean toward transactional consistency. Those who use CQRS may tend to lean toward eventual consistency. But which is correct? Frankly, neither of those leanings provide a domain-specific answer, only a technical preference. Is there a better way to break the tie?

Discussing this with Eric Evans revealed a very simple and sound guideline. When examining the use case (or story), ask whether it's the job of the user executing the use case to make the data consistent. If it is, try to make it transactionally consistent, but only by adhering to the other rules of **aggregate**. If it is another user's job, or the job of the system, allow it to be eventually consistent. That bit of wisdom not only provides a convenient tie breaker, it helps us gain a deeper understanding of our domain. It exposes the real system invariants: the ones that must be kept transactionally consistent. That understanding is much more valuable than defaulting to a technical leaning.

This is a great tip to add to **aggregate** rules of thumb. Since there are other forces to consider, it may not always lead to the final answer between transactional and eventual consistency, but will usually provide deeper insight into the model. This guideline is used later in Part III when the team revisits their **aggregate** boundaries.

Reasons To Break the Rules

An experienced DDD practitioner may at times decide to persist changes to multiple **aggregate** instances in a single transaction, but only with good reason. What might some reasons be? I discuss four reasons here. You may experience these and others.

Reason One: User Interface Convenience

Sometimes user interfaces, as a convenience, allow users to define the common characteristics of many things at once in order to create batches of them. Perhaps it happens frequently that team members want to create several backlog items as a batch. The user interface allows them to fill out all the common properties in one section, and then one-by-one the few distinguishing properties of each, eliminating repeated gestures. All of the new backlog items are then planned (created) at once:

```
public class ProductBacklogItemService ... {
    ...
    @Transactional
    public void planBatchOfProductBacklogItems(
        String aTenantId, String productId,
        BacklogItemDescription[] aDescriptions) {

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

        for (BacklogItemDescription desc : aDescriptions) {
            BacklogItem plannedBacklogItem =
                product.planBacklogItem(
                    desc.summary(),
                    desc.category(),
                    BacklogItemType.valueOf(
                            desc.backlogItemType()),
                    StoryPoints.valueOf(
                            desc.storyPoints()));

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

Does this cause a problem with managing invariants? In this case, no, since it would not matter whether these were created one at a time or in batch. The objects being instantiated are full **aggregates**, which themselves maintain their own invariants. Thus, if creating a batch of **aggregate** instances all at once is semantically no different than creating one at a time repeatedly, it represents one reason to break the rule of thumb with impunity.

Udi Dahan recommends avoiding the creation of special batch application services like the one above. Instead, a [Message Bus] would be used to batch multiple application service invocations together. This is done by defining a logical message type to represent a single invocation, with the client sending multiple logical messages together in the same physical message. On the server-side the [Message Bus] processes the physical message in a single transaction, delivering each logical message individually to a class which handles the “plan product backlog item message” for processing (equivalent in implementation to an application service method), all either succeeding or failing together.

Reason Two: Lack of Technical Mechanisms

Eventual consistency requires the use of some kind of out-of-band processing capability, such as messaging, timers, or background threads. What if the project you are working on has no provision for any such mechanism? While most of us would consider that strange, I have faced that very limitation. With no messaging mechanism, no background timers, and no other home-grown threading capabilities, what could be done?

If we aren't careful, this situation could lead us back toward designing large cluster **aggregates**. While that might make us feel like we are adhering to the single transaction rule, as previously discussed it would also degrade performance and limit scalability. To avoid that, perhaps we could instead change the system's **aggregates** altogether, forcing the model to solve our challenges. We've already considered the possibility that project specifications may be jealously guarded, leaving us little room for negotiating previously unimagined domain concepts. That's not really the DDD way, but sometimes it does happen. The conditions may allow for no reasonable way to alter the modeling circumstances in our favor. In such cases project dynamics may force us to modify two or more **aggregate** instances in one transaction. However obvious this might seem, such a decision should not be made too hastily.

Consider an additional factor that could further support diverting from the rule: *user-aggregate affinity*. Are the business work flows such that only one user would be focused on one set of **aggregate** instances at any given time? Ensuring user-aggregate affinity makes the decision to alter multiple **aggregate** instances in a single transaction more sound since it tends to prevent the violation of invariants and transactional collisions. Even with user-aggregate affinity, in rare situations users may face concurrency conflicts. Yet each **aggregate** would still be protected from that by using optimistic concurrency. Anyway, concurrency conflicts can happen in any system, and even more frequently when user-aggregate affinity is not our ally. Besides, recovering from concurrency conflicts is straightforward when encountered at rare times. Thus, when our design is forced to, sometimes it works out well to modify multiple **aggregate** instances in one transaction.

Reason Three: Global Transactions

Another influence considered is the effects of legacy technologies and enterprise policies. One such might be the need to strictly adhere to the use of global, two-phase commit transactions. This is one of those situations that may be impossible to push back on, at least in the short term.

Even if you must use a global transaction, you don't necessarily have to modify multiple **aggregate** instances at once in your local **bounded context**. If you can avoid doing so, at least you can prevent transactional contention in your **core domain** and actually obey the rules of **aggregates** as far as it depends on you. The downside to global transactions is that your system will probably never scale as it could if you were able to avoid two-phase commits and the immediate consistency that goes along with them.

Reason Four: Query Performance

There may be times when it's best to hold direct object references to other **aggregates**. This could be used to ease **repository** query performance issues. These must be weighed carefully in the light of potential size and overall performance trade-off implications. One example of breaking the rule of reference by identity is given in Part III.

Adhering to the Rules

You may experience user interface design decisions, technical limitations or stiff policies in your enterprise environment, or other factors, that require you to make some compromises. Certainly we don't go in search of excuses to break the **aggregate** rules of thumb. In the long run, adhering to the rules will benefit our projects. We'll have consistency where necessary, and support optimally performing and highly scalable systems.

Looking Forward to Part III

We are now resolved to design small **aggregates** that form boundaries around true business invariants, to prefer reference by identity between **aggregates**, and to use eventual consistency to manage cross-**aggregate** dependencies. How will adhering to these rules affect the design of our Scrum model? That's the focus of Part III. We'll see how the project team rethinks their design again, applying their new-found techniques.

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

[DDD] Eric Evans; *Domain-Driven Design—Tackling Complexity in the Heart of Software*.

[Dahan] Udi Dahan; *Clarified CQRS*; http://www.udidahan.com/2009/12/09/clarified-cqrs/

[Fowler] Martin Fowler; *CQRS*; http://martinfowler.com/bliki/CQRS.html

[GoF] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides; *Design Patterns: Elements of Reusable Object-Oriented Software;* see the Observer pattern.

[Helland] Pat Helland; *Life beyond Distributed Transactions: an Apostate's Opinion*; http://www.ics.uci.edu/~cs223/papers/cidr07p15.pdf

[Message Bus] *NServiceBus* is a framework that supports this pattern; http://www.nservicebus.com/

[POSA1] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad; *Pattern-Oriented Software Architecture Volume 1: A System of Patterns;* see the Publisher-Subscriber pattern.

[Young] Greg Young; *CQRS and Event Sourcing*; http://codebetter.com/gregyoung/2010/02/13/cqrs-and-event-sourcing/

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

Effective Aggregate Design Part III: Gaining Insight Through Discovery

Vaughn Vernon: vvernon@shiftmethod.com

Follow on Twitter: @VaughnVernon

Part II discussed how [DDD] **aggregates** reference other **aggregates**, and how to leverage eventual consistency to keep separate **aggregate** instances in harmony. In Part III we'll see how adhering to the rules of **aggregate** affects the design of a Scrum model. We'll see how the project team rethinks their design again, applying new-found techniques. That effort leads to the discovery of new insights into the model. Their various ideas are tried and then superseded.

Rethinking the Design, Again

After the refactoring iteration that broke up the large cluster `Product`, the `BacklogItem` now stands alone as its own **aggregate**. It reflects the model presented in Figure 7. The team composed a collection of `Task` instances inside the `BacklogItem` **aggregate**. Each `BacklogItem` has a globally unique identity, its `BacklogItemId`. All associations to other **aggregates** are inferred through identities. That means its parent `Product`, the `Release` it is scheduled within, and the `Sprint` to which it is committed, are referenced by identities. It seems fairly small. With the team now jazzed about designing small **aggregates**, could they possibly overdo it in that direction?

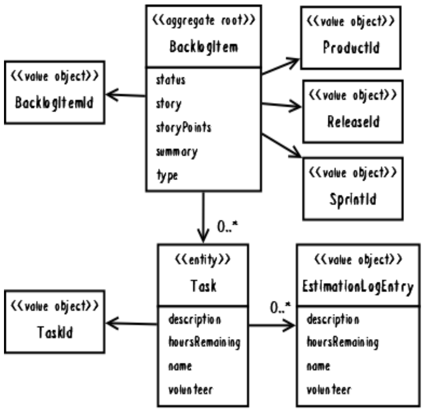

Figure 7: The fully composed `BacklogItem` **aggregate**.

Despite the good feeling coming out of that previous iteration, there is still some concern. For example, the `story` attribute allows for a good deal of text. Teams developing agile stories won't write lengthy prose. Even so, there is an optional editor component that supports writing rich use case definitions. Those could be many thousands of bytes. It's worth considering the possible overhead.

Given this potential overhead and the errors already made in designing the large cluster `Product` of Figures 1 and 3 in Part I, the team is now on a mission to reduce the size of every **aggregate** in the **bounded context**. Crucial questions arise. Is there a true invariant between `BacklogItem` and `Task` that this relationship must maintain? Or is this yet another case where the association can be further broken apart, with two separate **aggregates** being safely formed? What is the total cost of keeping the design as it is?

A key to making a proper determination lies in the **ubiquitous language**. Here is where an invariant is stated:

- When progress is made on a backlog item task, the team member will estimate task hours remaining.
- When a team member estimates that zero hours are remaining on a specific task, the backlog item checks all tasks for any remaining hours. If no hours remain on any tasks, the backlog item status is automatically changed to done.
- When a team member estimates that one or more hours are remaining on a specific task and the backlog item's status is already done, the status is automatically regressed.

This sure seems like a true invariant. The backlog item's correct status is automatically adjusted and completely dependent on the total number of hours remaining on all its tasks. If the total number of task hours and the backlog item status are to remain consistent, it seems as if Figure 7 does stipulate the correct **aggregate** consistency boundary. However, the team should still determine what the current cluster could cost in terms of performance and scalability. That would be weighed against what they might save if the backlog item status could be eventually consistent with the total task hours remaining.

Some will see this as a classic opportunity to use eventual consistency, but we won't jump to that conclusion just yet. Let's analyze a transactional consistency approach, then investigate what could be accomplished using eventual consistency. We can then each draw our own conclusion as to which approach is preferred.

Estimating Aggregate Cost

As Figure 7 shows, each `Task` holds a collection of a series of `EstimationLogEntry` instances. These logs model the specific occasions when a team member enters a new estimation of hours remaining. In practical terms, how many `Task` elements will each `BacklogItem` hold, and how many `EstimationLogEntry` elements will a given `Task` hold? It's hard to say exactly. It's largely a measure of how complex any one task is and how long a sprint lasts. But some back-of-the-envelope calculations (BOTE) might help [Pearls].

Task hours are usually re-estimated each day after a team member works on a given task. Let's say that most sprints are either two or three weeks in length. There will be longer sprints, but a two-to-three-week timespan is common enough. So let's select a number of days somewhere in between 10 days and 15 days. Without being too precise, 12 days works well since there may actually be more two-week than three-week sprints.

Next consider the number of hours assigned to each task. Remembering that since tasks must be broken down into manageable units, we generally use a number of hours between 4 and 16. Normally if a task exceeds a 12-hour estimate, Scrum experts suggest breaking it down further. But using 12 hours as a first test makes it easier to simulate work evenly. We can say that tasks are worked on for one hour each of the 12 days of the sprint. Doing so favors more complex tasks. So we'll figure 12 re-estimations per task, assuming that each task starts out with 12 hours allocated to it.

The question remains: How many tasks would be required per backlog item? That too is a difficult question to answer. What if we thought in terms of there being two or three tasks required per **layer** or **hexagonal port-adapter** [Cockburn] for a given feature slice? For example, we might count three for **user interface layer**, two for the **application layer**, three for the **domain layer**, and three for the **infrastructure layer**. That would bring us to 11 total tasks. It might be just right or a bit slim, but we've already erred on the side of numerous task estimations. Let's bump it up to 12 tasks per backlog item to be more liberal. With that we are allowing for 12 tasks, each with 12 estimation logs, or *144 total collected objects per backlog item*. While this may be more than the norm, it gives us a chunky BOTE calculation to work with.

There is another variable to be considered. If Scrum expert advice to define smaller tasks is commonly followed, it would change things somewhat. Doubling the number of tasks (24) and halving the number of estimation log entries (6) would still produce 144 total objects. However, it would cause more tasks to be loaded (24 rather than 12) during all estimation requests, consuming more memory on each. The team will try various combinations to see if there was any significant impact on their performance tests. But to start they will use 12 tasks of 12 hours each.

Common Usage Scenarios

Now it's important to consider common usage scenarios. How often will one user request need to load all 144 objects into memory at once? Would that ever happen? It seems not, but they need to check. If not, what's the likely high end count of objects? Also, will there typically be multi-client usage that causes concurrency contention on backlog items? Let's see.

The following scenarios are based on the use of Hibernate for persistence. Also, each **entity** type has its own optimistic concurrency version attribute. This is workable because the changing status invariant is managed on the `BacklogItem` **root entity**. When the status is automatically altered (to done or back to committed) the **root's** version is bumped. Thus, changes to tasks can happen independently of each other and without impacting the **root** each time one is modified, unless it results in a status change. (The following analysis could need to be revisited if using, for example, document-based storage, since the **root** is effectively modified every time a collected part is modified.)

When a backlog item is first created, there are zero contained tasks. Normally it is not until sprint planning that tasks are defined. During that meeting tasks are identified by the team. As each one is called out, a team member adds it to the corresponding backlog item. There is no need for two team members to contend with each other for the **aggregate**, as if racing to see who can enter new tasks the quickest. That would cause collision and one of the two requests would fail (for the same reason adding various parts to `Product` simultaneously previously failed). However, the two team members would probably soon figure out how counterproductive their redundant work is.

If the developers learned that multiple users do indeed regularly want to add tasks together, it would change the analysis significantly. That understanding could immediately tip the scales in favor of breaking `BacklogItem` and `Task` into two separate **aggregates**. On the other hand, this could also be a perfect time to tune the Hibernate mapping by setting `optimistic-lock` option to `false`. Allowing tasks to grow simultaneously could make sense in this case, especially if they don't pose performance and scalability issues.

If tasks are at first estimated at zero hours and later updated to an accurate estimate, we still don't tend to experience concurrency contention, although this would add one additional estimation log entry, pushing our BOTE to 13 total. Simultaneous use here does not change the backlog item status. Again, it only advances to done by going from greater-than zero to zero hours, or regresses to committed if already done and hours are changed from zero to one or more—two uncommon events.

Will daily estimations cause problems? On day one of the sprint there are usually zero estimation logs on a given task of a backlog item. At the end of day one, each volunteer team member working on a task reduces the estimated hours by one. This adds a new estimation log to each task, but the backlog item's status remains unaffected. There is never contention on a task because just one team member adjusts its hours. It's not until day 12 that we reach the point of status transition. Still, as each of any 11 tasks are reduced to zero hours, the backlog item's status is not altered. It's only the very last estimation, the 144th on the 12th task, that causes automatic status transition to the done state.

This analysis has led the team to an important realization. Even if they alter the usage scenarios, accelerating task completion by double (six days), or even mixing it up completely, it doesn't change anything. It's always the final estimation that transitions the status, which modifies the **root**. This seems like a safe design, although memory overhead is still in question.

Memory Consumption

Now to address the memory consumption. Important here is that estimations are logged by date as **value objects**. If a team member re-estimates any number of times on a single day, only the most recent estimation is retained. The latest **value** of the same date replaces the previous one in the collection. At this point there's no requirement to track task estimation mistakes. There is the assumption that a task will never have more estimation log entries than the number of days the sprint is in progress. That assumption changes if tasks were defined one or more days before the sprint planning meeting, and hours were re-estimated on any of those earlier days. There would be one extra log for each day that occurred.

What about the total number of tasks and estimations in memory for each re-estimation? When using lazy loading for the tasks and estimation logs, we would have as many as 12 plus 12 collected objects in memory at one time per request. This is because all 12 tasks would be loaded when accessing that collection. To add the latest estimation log entry to one of those tasks, we'd have to load the collection of estimation log entries. That would be up to another 12 objects. In the end the **aggregate** design requires one backlog item, 12 tasks, and 12 log entries, or 25 objects maximum total. That's not very many; it's a small **aggregate**. Another factor is that the higher end of objects (e.g. 25) is not reached until the last day of the sprint. During much of the sprint the **aggregate** is even smaller.

Will this design cause performance problems because of lazy loads? Possibly, because it actually requires two lazy loads, one for the tasks and one for the estimation log entries for one of the tasks. The team will have to test to investigate the possible overhead of the multiple fetches.

There's another factor. Scrum enables teams to experiment in order to identity the right planning model for their practices. As explained in [Story Points], experienced teams with a well-known velocity can estimate using story points rather than task hours. As they define each task, they can assign just one hour to each task. During the sprint they will re-estimate only once per task, changing one hour to zero when the task is completed. As it pertains to **aggregate** design, using story points reduces the total number of estimation logs per task to just one, and almost eliminates memory overhead. Later on, *ProjectOvation* developers will be able to analytically determine (on average) how many actual tasks and estimation log entries exist per backlog item by examining real production data.

The forgoing analysis was enough to motivate the team to test against their BOTE calculations. After inconclusive results, however, they decide that there were still too many variables to be confident that this design deals well with their concerns. There were enough unknowns to consider an alternative design.

Exploring Another Alternative Design

To be thorough, the team wants to think through what they would have to do to make `Task` an independent **aggregate**, and if that would actually work to their benefit. What they envision is seen in Figure 8. Doing this would reduce part composition overhead by 12 objects and reduce lazy load overhead. In fact, this design gives them the option to eagerly load estimation log entries in all cases if that would perform best.

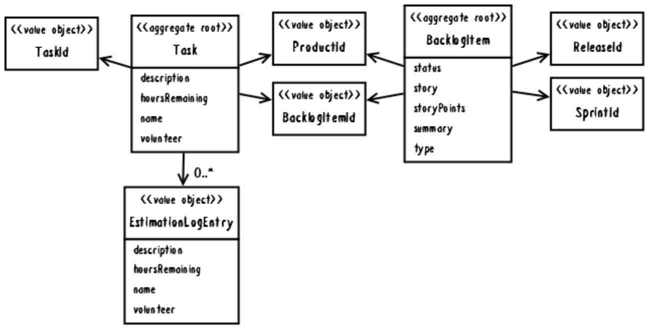

Figure 8: `BacklogItem` and `Task` modeled as separate **aggregates**.

The developers agree not to modify separate **aggregates**, both the `Task` and the `BacklogItem`, in the same transaction. They must determine if they can perform a necessary automatic status change within an acceptable time frame. They'd be weakening the invariant's consistency since the status can't be consistent by transaction. Would that be acceptable? They discuss the matter with the domain experts and learn that some delay between the final zero-hour estimate and the status being set to done, and visa versa, would be acceptable.

Implementing Eventual Consistency

Here is how it could work. When a `Task` processes an `estimateHoursRemaining()` command it publishes a corresponding **domain event**. It does that already, but the team would now leverage the **event** to achieve eventual consistency. The **event** is modeled with the following properties:

```
public class TaskHoursRemainingEstimated implements DomainEvent {
    private Date occurredOn;
    private TenantId tenantId;
    private BacklogItemId backlogItemId;
    private TaskId taskId;
    private int hoursRemaining;
    ...
}
```

A specialized subscriber would now listen for these and delegate to a **domain service** to coordinate the consistency processing. The **service** would:

- Use the `BacklogItemRepository` to retrieve the identified `BacklogItem`.
- Use the `TaskRepository` to retrieve all `Task` instances associated with the identified `BacklogItem`.
- Execute the `BacklogItem` command named `estimateTaskHoursRemaining()` passing the **domain event's** `hoursRemaining` and the retrieved `Task` instances. The `BacklogItem` may transition its status depending on parameters.

The team should find a way to optimize this. The three-step design requires all `Task` instances to be loaded every time a re-estimation occurs. When using our BOTE and advancing continuously toward done, 143 out of 144 times that's unnecessary. This could be optimized this pretty easily. Instead of using the **repository** to get all `Task` instances, they could simply ask it for the sum of all `Task` hours as calculated by the database:

```
public class TaskRepositoryImpl implements TaskRepository {
    ...
    public int totalBacklogItemTaskHoursRemaining(
            TenantId aTenantId,
            BacklogItemId aBacklogItemId) {

        Query query = session.createQuery(
            "select sum(task.hoursRemaining) from Task task "
            + "where task.tenantId = ? and "
            + "task.backlogItemId = ?");
        ...
    }
}
```

Eventual consistency complicates the user interface a bit. Unless the status transition can be achieved within a few hundred milliseconds, how would the user interface display the new state? Should they place business logic in the view to determine the current status? That would constitute a smart UI anti-pattern. Perhaps the view would just display the stale status and allow users to deal with the visual inconsistency. That could easily be perceived as a bug, or at least be pretty annoying.

The view could use a background Ajax polling request, but that could be quite inefficient. Since the view component could not easily determine exactly when checking for a status update is necessary, most Ajax pings would be unnecessary. Using our BOTE numbers, 143 of 144 re-estimations would not cause the status update, which is a lot of redundant requests on the web tier. With the right server-side support the clients could instead depend on Comet (a.k.a. Ajax Push). Although a nice challenge, that introduces a completely new technology that the team has no experience using.

On the other hand, perhaps the best solution is the simplest. They could opt to place a visual cue on the screen that informs the user that the current status is uncertain. The view could suggest a time frame for checking back or refreshing. Alternatively, the changed status will probably show on the next rendered view. That's safe. The team would need to run some user acceptance tests, but it looks hopeful.

Is It the Team Member's Job?

One important question has thus far been completely overlooked. Whose job is it to bring a backlog item's status into consistency with all remaining task hours? Does a team member using Scrum care if the parent backlog item's status transitions to done just as they set the last task's hours to zero? Will they always know they are working with the last task that has remaining hours? Perhaps they will and perhaps it is the responsibility of each team member to bring each backlog item to official completion.

On the other hand, what if there is ever another project stakeholder involved? For example, the product owner or some other person may desire to check the candidate backlog item for satisfactory completion. Maybe they want to use the feature on a continuous integration server first. If they are happy with the developers' claim of completion, they will manually mark the status as done. This certainly changes the game, indicating that neither transactional nor eventual consistency is necessary. Tasks could be split off from their parent backlog item because this new use case allows it. However, if it is really the team members that should cause the automatic transition to done, it would mean that tasks should probably be composed within the backlog item to allow for transactional consistency. Interestingly, there is no clear answer here either, which probably indicates that it should be an optional application preference. Leaving tasks within their backlog item solves the consistency problem, and it's a modeling choice that can support both automatic or manual status transitions.

This valuable exercise has uncovered a completely new aspect of the domain. It seems like teams should be able to configure a work flow preference. They aren't going to implement such a feature now, but they will promote it for further discussion. Asking 'whose job is it?' led them to a few vital perceptions about their domain.

Next, one of the developers made a very practical suggestion as an alternative to this whole analysis. If they are chiefly concerned with the possible overhead of the `story` attribute, why not do something about that specifically? They could reduce the total storage capacity for the `story` and in addition create a new `useCaseDefinition` property too. They could design it to lazy load, since much of the time it would never be used. Or they could even design it as a separate **aggregate**, only loading it when needed. With that idea they realized this could be a good time to break the rule to reference external **aggregates** only by identity. It seems like a suitable modeling choice to use a direct object reference, and declare its object-relational mapping so as to lazily load it. Perhaps that makes sense.

Time for Decisions

Based on all this analysis, currently the team is shying away from splitting `Task` from `BacklogItem`. They can't be certain that splitting it now is worth the extra effort, the risk of leaving the true invariant unprotected, or allowing users to experience a possible stale status in the view. The current **aggregate**, as they understand it, is fairly small as is. Even if their common worse case loaded 50 objects rather than 25, it's still a reasonably sized cluster. *For now they will plan around the specialized use case definition holder*. Doing that is a quick win with lots of benefits. It adds little risk, because it will work now, and in the future if they decide to split `Task` from `BacklogItem`.

The option to split it in two remains in their hip pocket just in case. After further experimentation with the current design, running it through performance and load tests, as well investigating user acceptance with an eventually consistent status, it will become more clear which approach is best. The BOTE numbers could prove to be wrong if in production the **aggregate** is larger than imagined. If so, the team will no doubt split it into two.

If you were a member of the *ProjectOvation* team, which modeling option would you have chosen?

Summary

Don't shy away from discovery sessions as demonstrated above. That entire effort would require 30 minutes, and perhaps as much as 60 minutes at worse case. It's well worth the time to gain deeper insight into your **core domain**.

Using a real-world example domain model, we have examined how crucial it is to follow the rules of thumb when designing **aggregates**:

- Model True Invariants In Consistency Boundaries
- Design Small Aggregates
- Reference Other Aggregates By Identity
- Use Eventual Consistency Outside the Boundary (after asking whose job it is)

If we adhere to the rules, we'll have consistency where necessary, and support optimally performing and highly scalable systems, all while capturing the **ubiquitous language** of our business domain in a carefully crafted model.

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

[Cockburn] Alistair Cockburn; Hexagonal Architecture; http://alistair.cockburn.us/Hexagonal+architecture

[DDD] Eric Evans; *Domain-Driven Design—Tackling Complexity in the Heart of Software*.

[Pearls] Jon Bentley; *Programming Pearls, Second Edition;* http://cs.bell-labs.com/cm/cs/pearls/bote.html

[Story Points] Jeff Sutherland; *Story Points: Why are they better than hours?*; http://scrum.jeffsutherland.com/2010/04/story-points-why-are-they-better-than.html

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/; you can reach him by email here: vvernon@shiftmethod.com; and follow him on Twitter here: @VaughnVernon