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.

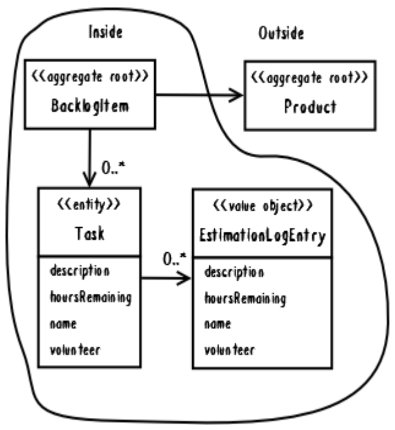

Figure 5: There are two **aggregates**, not one.

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?

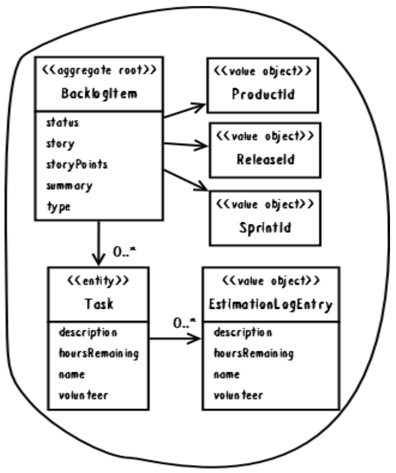

Figure 6: The `BacklogItem` **aggregate**, inferring associations outside its boundary with identities.

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