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 Their various ideas are tried and then superseded.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?

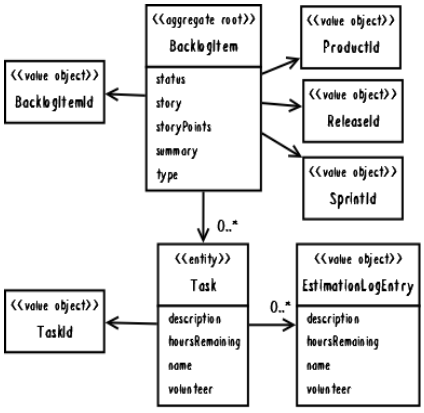

Figure 7: The fully composed `BacklogItem` **aggregate**.

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.

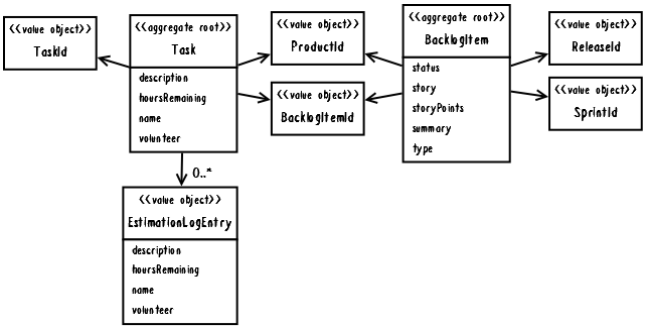

Figure 8: `BacklogItem` and `Task` modeled as separate **aggregates**.

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