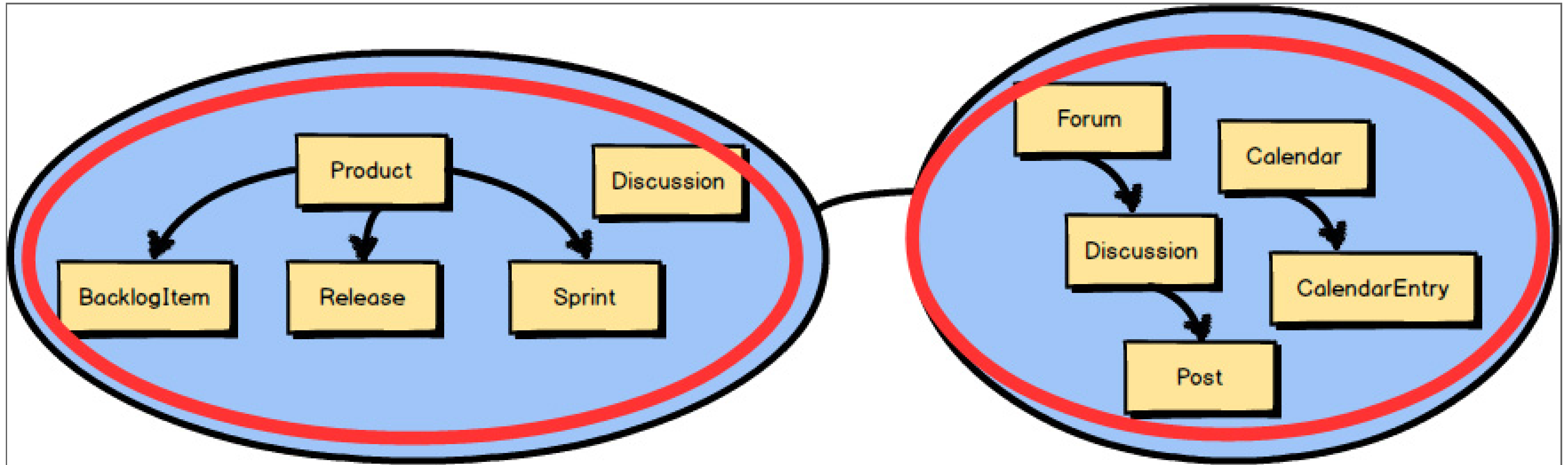
Software Architectures

Domain Driven Design

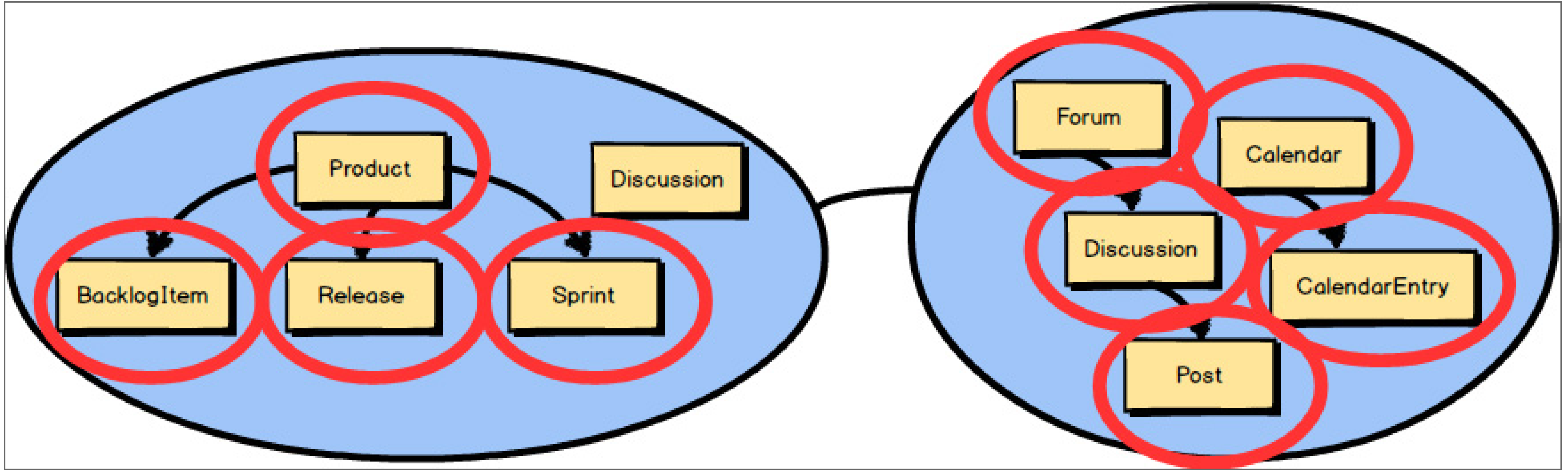
Content

1. Tactical Design with Aggregates
2. Tactical Design with Events

Tactical Design with Aggregates


Product
Discussion
BacklogItem
Release
Sprint
Forum
Calendar
Discussion
CalendarEntry
Post

Aggregates

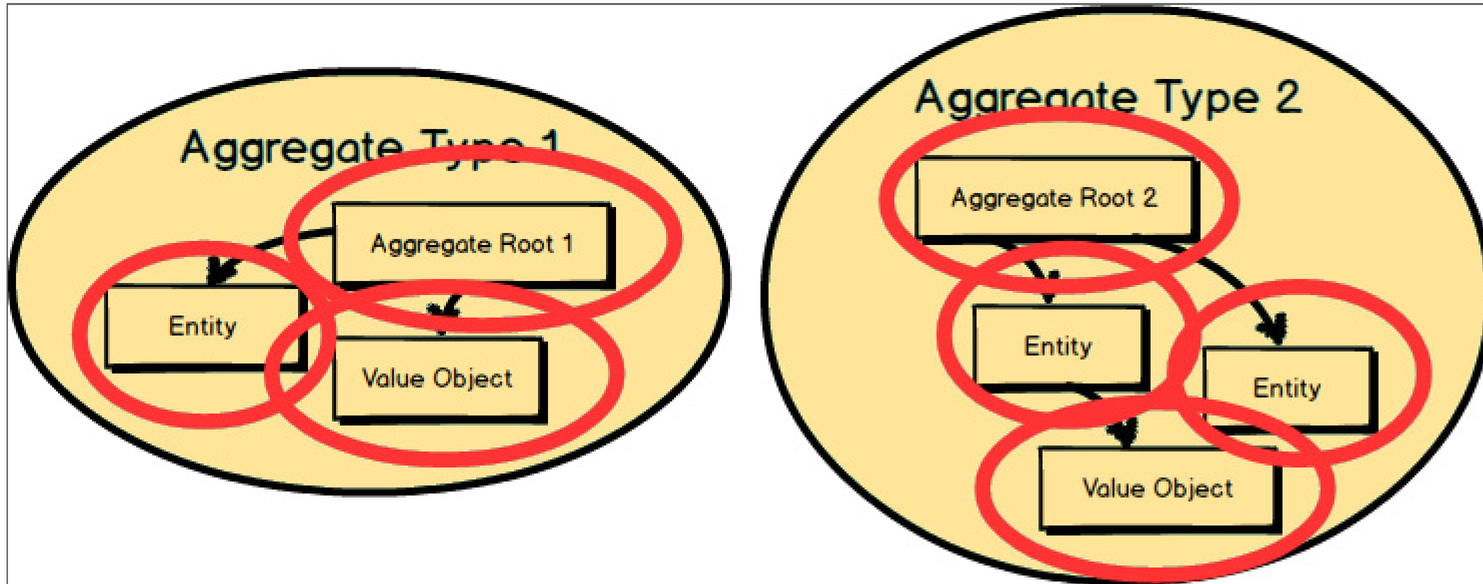


- each bounded context in an aggregate
- discussion is value object

Entity

- An Entity models an individual thing.
- Each Entity has a unique identity in that you can distinguish its individuality from among all other Entities of the same or a different type.
- Most of the times an Entity will be mutable = state will change in time

Aggregate



- Each Aggregate is composed of one or more Entities
- One Entity is called the Aggregate Root
- Aggregates may also have Value Objects composed on them

Value Object

- models an immutable conceptual whole.
- does not have a unique identity, and equivalence is determined by comparing the attributes encapsulated by the Value type.
- often used to describe, quantify, or measure an Entity.

Value Object vs Entity

Identity:

Entity:

- have a distinct identity that is usually represented by a unique identifier (ID)
- are defined by their identity, and two entities with the same attributes but different IDs are considered different entities.

Value Object:

- do not have a distinct identity.
- they are defined solely by their attributes

Value Object vs Entity

Mutability:

Entity:

- entities are mutable, their state can change over time while maintaining the same identity.

Value Object:

- value Objects are immutable, meaning their state cannot change once they are created. If you need to modify a Value Object, you create a new one with the desired state.

Value Object vs Entity

Equality:

Entity: Equality for entities is based on their identity. If two entities have the same ID, they are considered equal.
Value Object: Equality for Value Objects is based on the equality of their attributes.

Value Object vs Entity

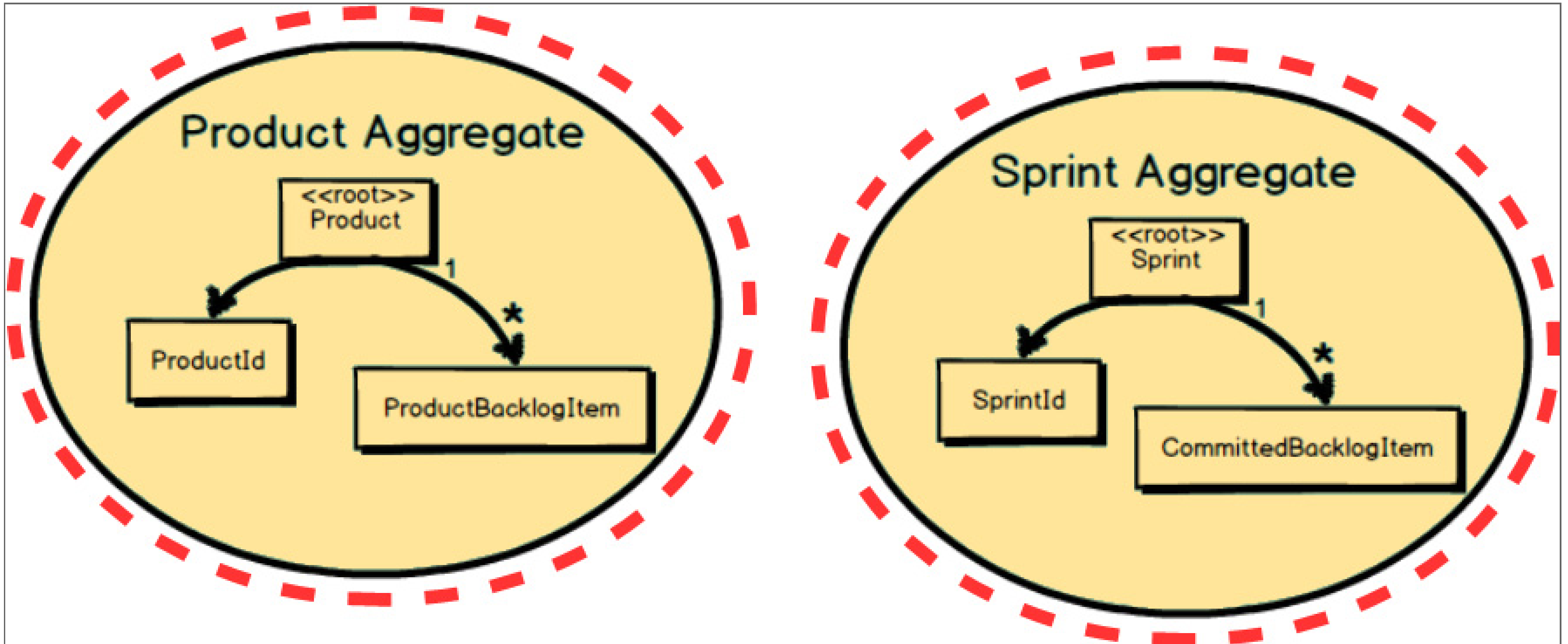
Lifecycle:

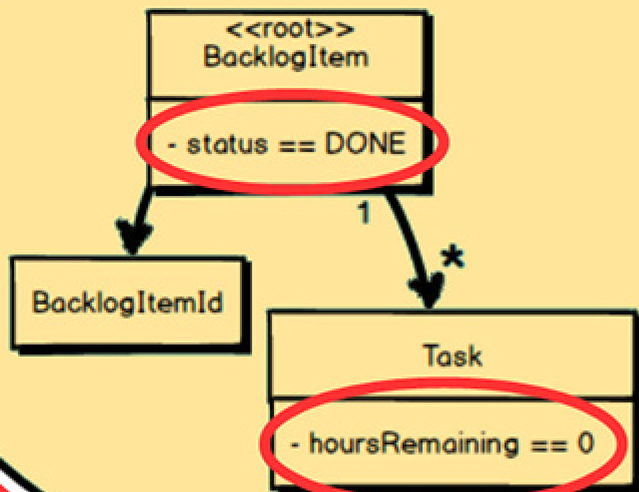
Entity: Entities have a longer lifecycle and represent concepts that persist over time. They are typically stored in a database and can be retrieved and updated as needed.
Value Object: Value Objects have a shorter lifecycle and are often used to represent immutable values that are part of an Entity's state. They are not typically stored in a database as separate entities.

Four Rules of Thumb

1. Protect business invariants inside Aggregate boundaries.
2. Design small Aggregates.
3. Reference other Aggregates by identity only.
4. Update other Aggregates using eventual consistency.

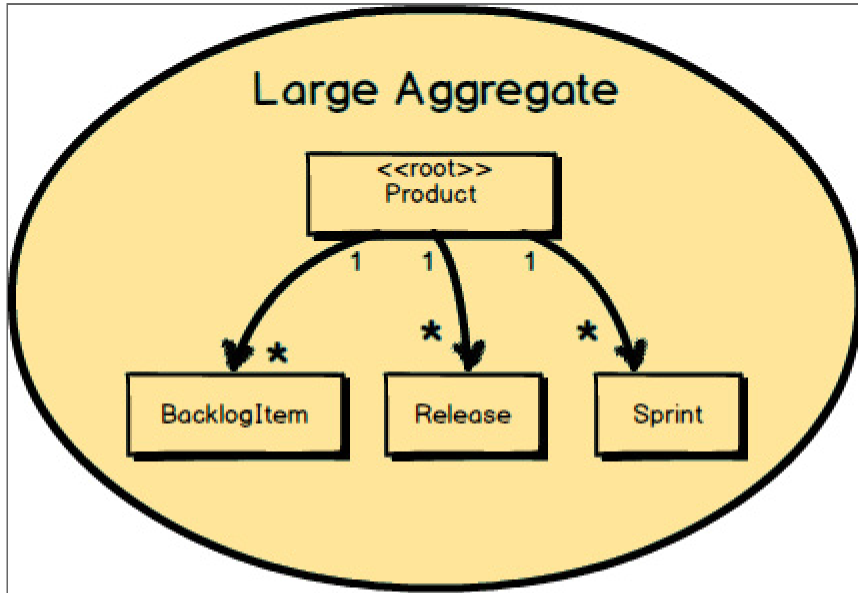
Rule 1: Protect Business Invariants




BacklogItem Aggregate
<<root>>
BacklogItem
- status == DONE
1
*
BacklogItemId
Task
- hoursRemaining == 0

There is a business rule that states, “When all Task instances have hoursRemaining of zero, the BacklogItem status must be set to DONE.” Thus, at the end of a transaction this very specific business invariant must be met. The business requires it.

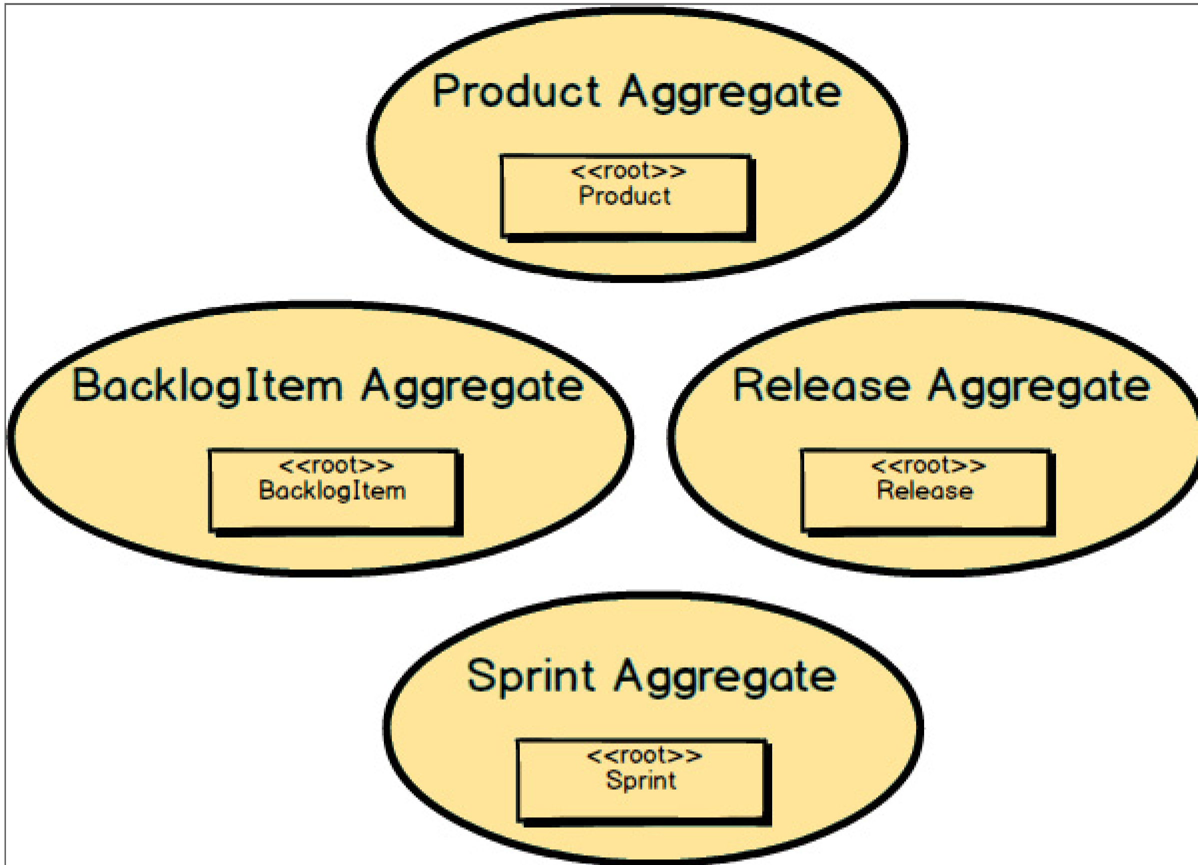
Rule 2: Design Small Aggregates



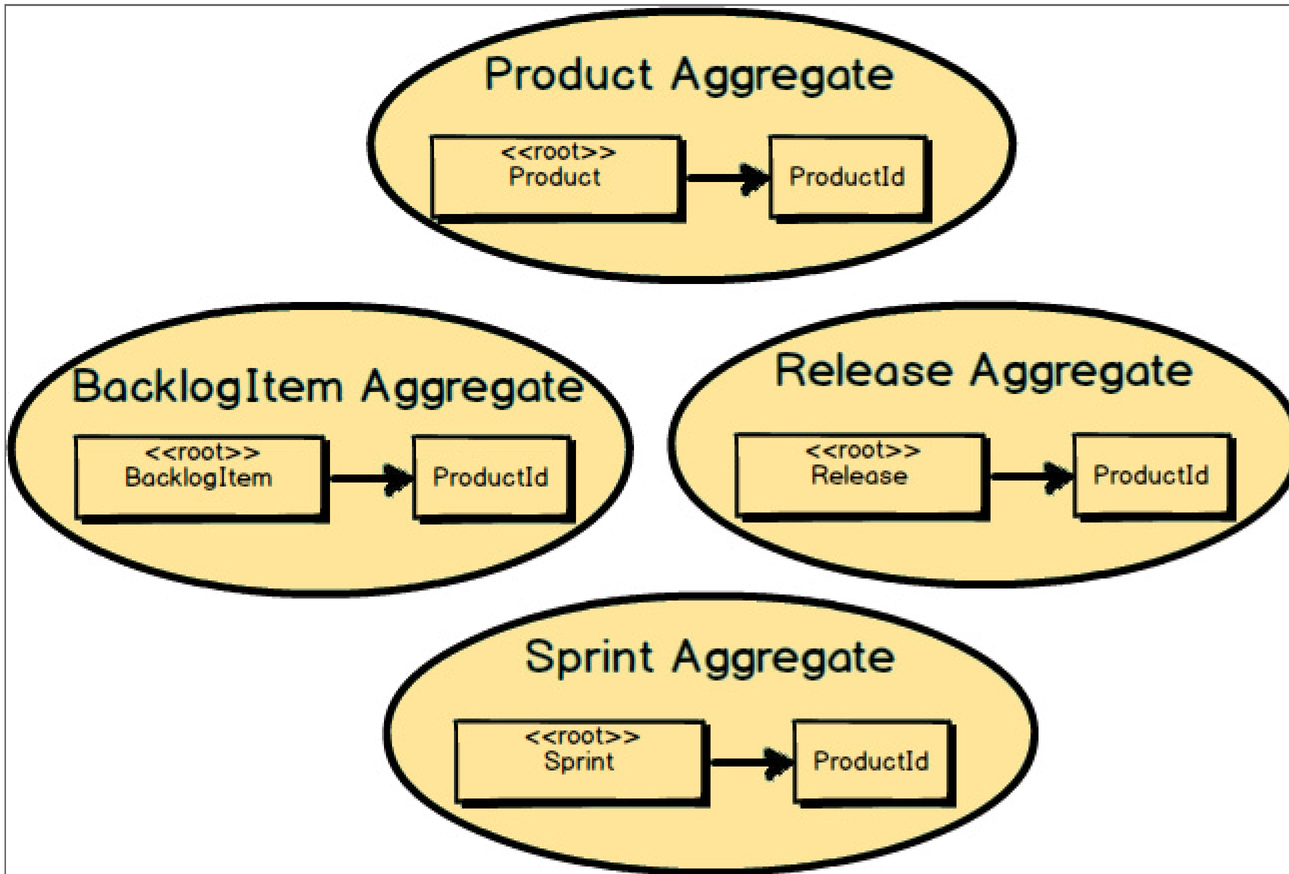
- In the preceding diagram the Aggregate that is represented is not small.
- Product literally contains a potentially very large collection of BacklogItem instances, a large collection of Release instances, and a large collection of Sprint instances.
- Over time, these collections could grow to be quite large, with thousands of BacklogItem instances and probably hundreds of Release and Sprint instances.
- This design approach is generally a very poor choice.

Small aggregates

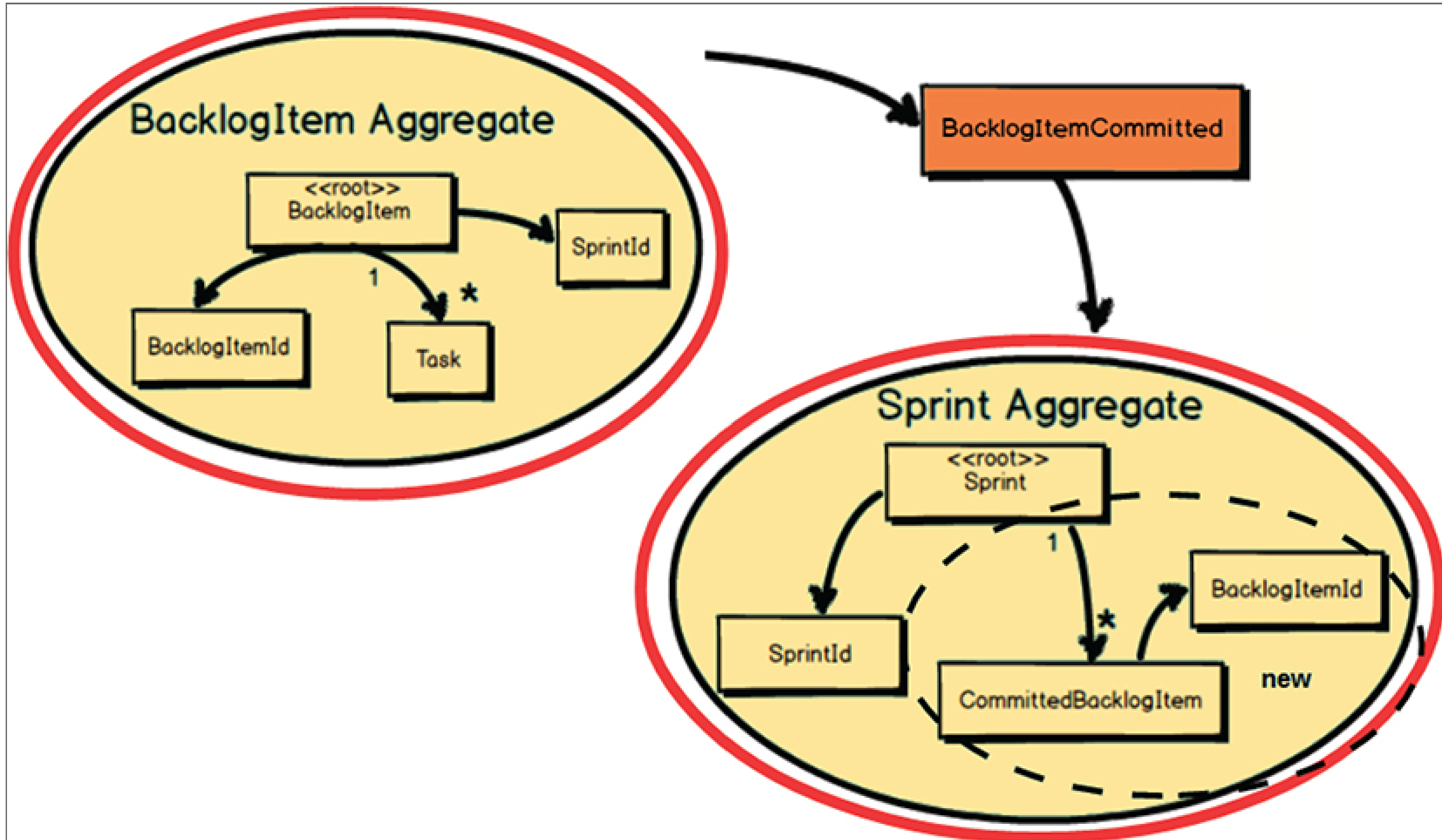
- load quickly, take less memory, and are faster to garbage collect

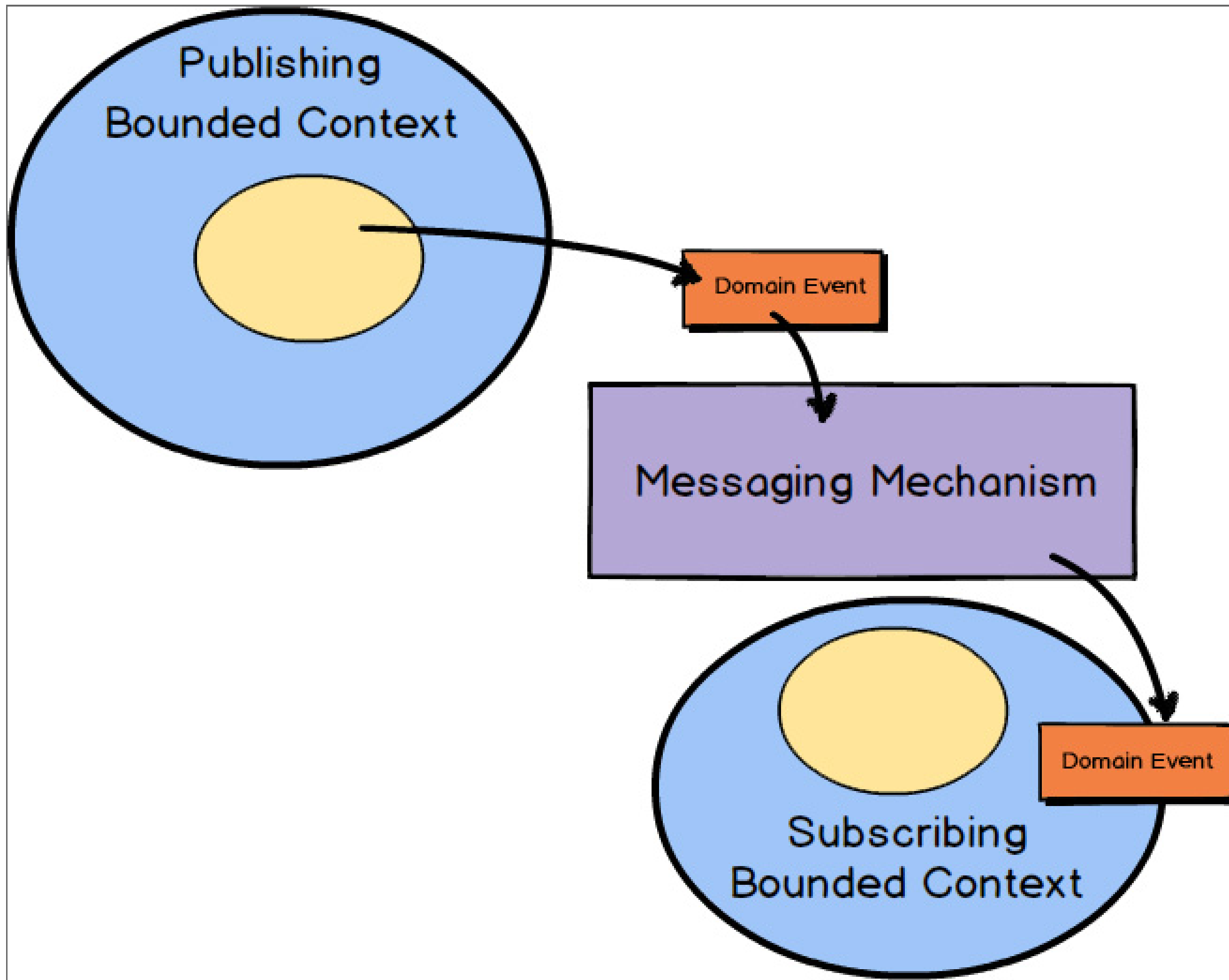


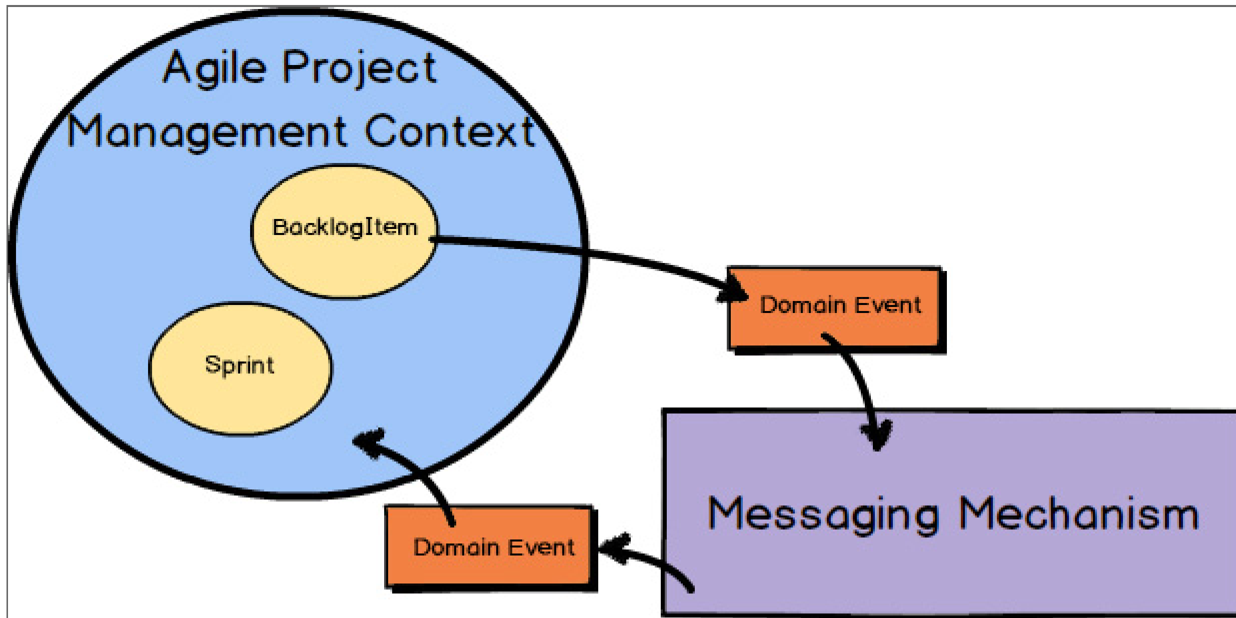
Rule 3: Reference Other Aggregates by Identity Only

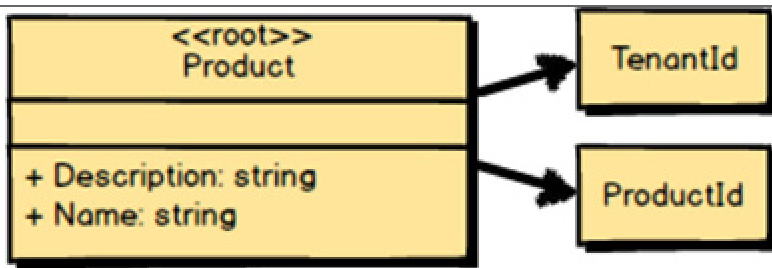


Rule 4: Update Other Aggregates Using Eventual Consistency



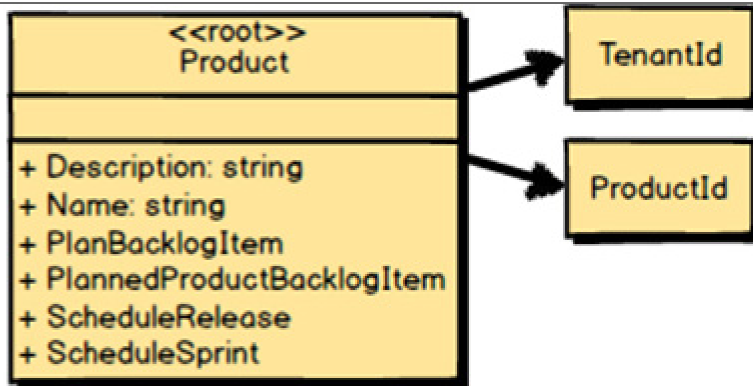

Publishing
Bounded Context
Domain Event
Messaging Mechanism
Domain Event
Subscribing
Bounded Context


Agile Project Management Context
BacklogItem
Sprint
Domain Event
Messaging Mechanism
Domain Event




```
public class Product : Entity
{
   ...
   public string Description
      { get; private set; }

   public string Name
      { get; private set; }
}
```




```
public class Product : Entity
{
  ...
  public void PlannedProductBacklogItem(...)
  {
    ...
  }
}
```

Core Domain: Scrum Project Management

Ubiquitous Language

- Product
- BacklogItem
- Release
- Sprint

Problems with Wrong Abstractions

- Ignores Ubiquitous Language
- Hard to model details of specific types
- Special cases and complex class hierarchy
- More code than necessary if modeling explicitly
- Will influence the user interface negatively
- Waste time and money pursuing wrong design
- Imagined future proofing your design will meet with failure when future concepts realized

Model Explicitly Per Ubiquitous Language

- Adheres to the mental model of Domain Experts
- Creates an understandable model
- Protects the organization's software investment
- Saves time and money

Modeling Steps

1. Start with Rule 2, Design Small Aggregates
2. Next use Rule 1, Protect Business Invariants. Make a chart with Aggregate names and list dependents under each
3. Ask Domain Experts for an acceptable time frame for updates to each dependent, for (a) immediate, (b) eventually (e.g. N seconds)
4. House all 3a components under one Aggregate
5. Plan to update all 3b dependents eventually

Unit Test Aggregates

1. Using Rule 2, Design Small Aggregates, will help make Aggregates testable
2. Unit tests are different from acceptance tests
3. Test for correctness and robustness of each Component

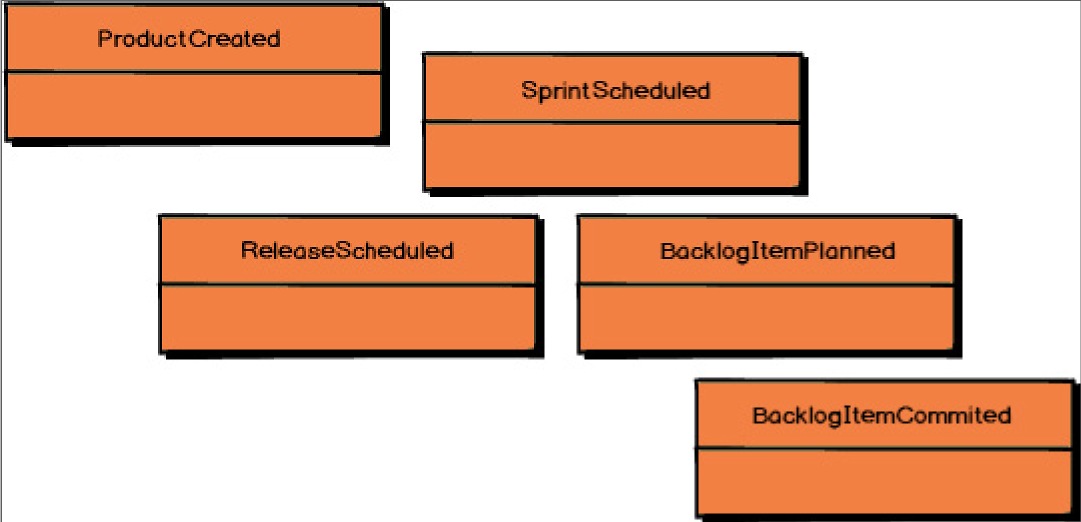
Tactical Design with Domain Events

Causal Consistency: An Example

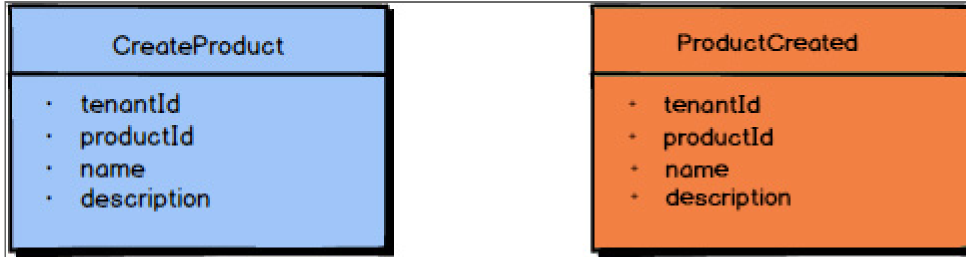
1. Sue posts a message saying, "I lost my wallet!"
2. Gary says in reply, "That's terrible!"
3. Sue posts a message saying, "Don't worry, I found my wallet!"
4. Gary replies, "That's great!"

Designing, Implementing, and Using Domain Events

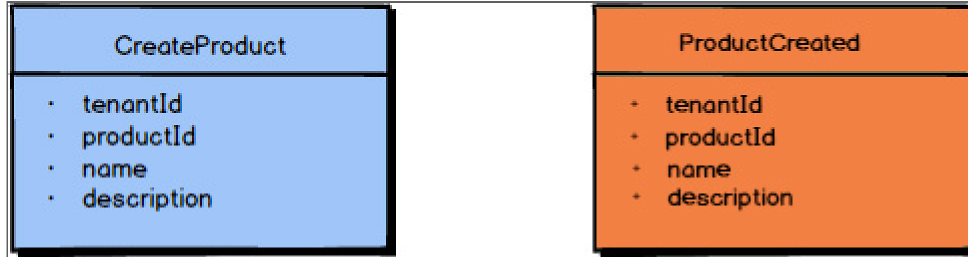
```
public interface DomainEvent
{
  public Date OccurredOn
  {
    get;
  }
}
```


ProductCreated
SprintScheduled
ReleaseScheduled
BacklogItemPlanned
BacklogItemCommited

Command vs event



Scrum domain events



ProductCreated

- tenantId
- productId
- name
- description

SprintScheduled

- tenantId
- sprintId
- productId
- name
- description
- startsOn
- endsOn

ReleaseScheduled

- tenantId
- releaseId
- productId
- name
- description
- targetDate

BacklogItemPlanned

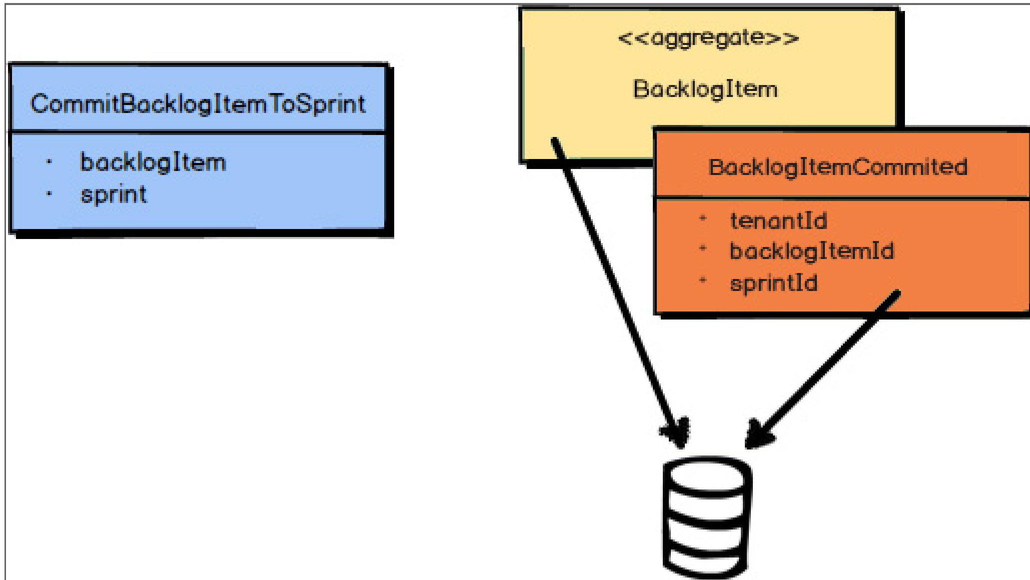
- tenantId
- backlogItemId
- productId
- sprintId
- story
- summary

BacklogItemCommited

- tenantId
- backlogItemId
- sprintId

Save to DB

- both aggregate and the event (event log)

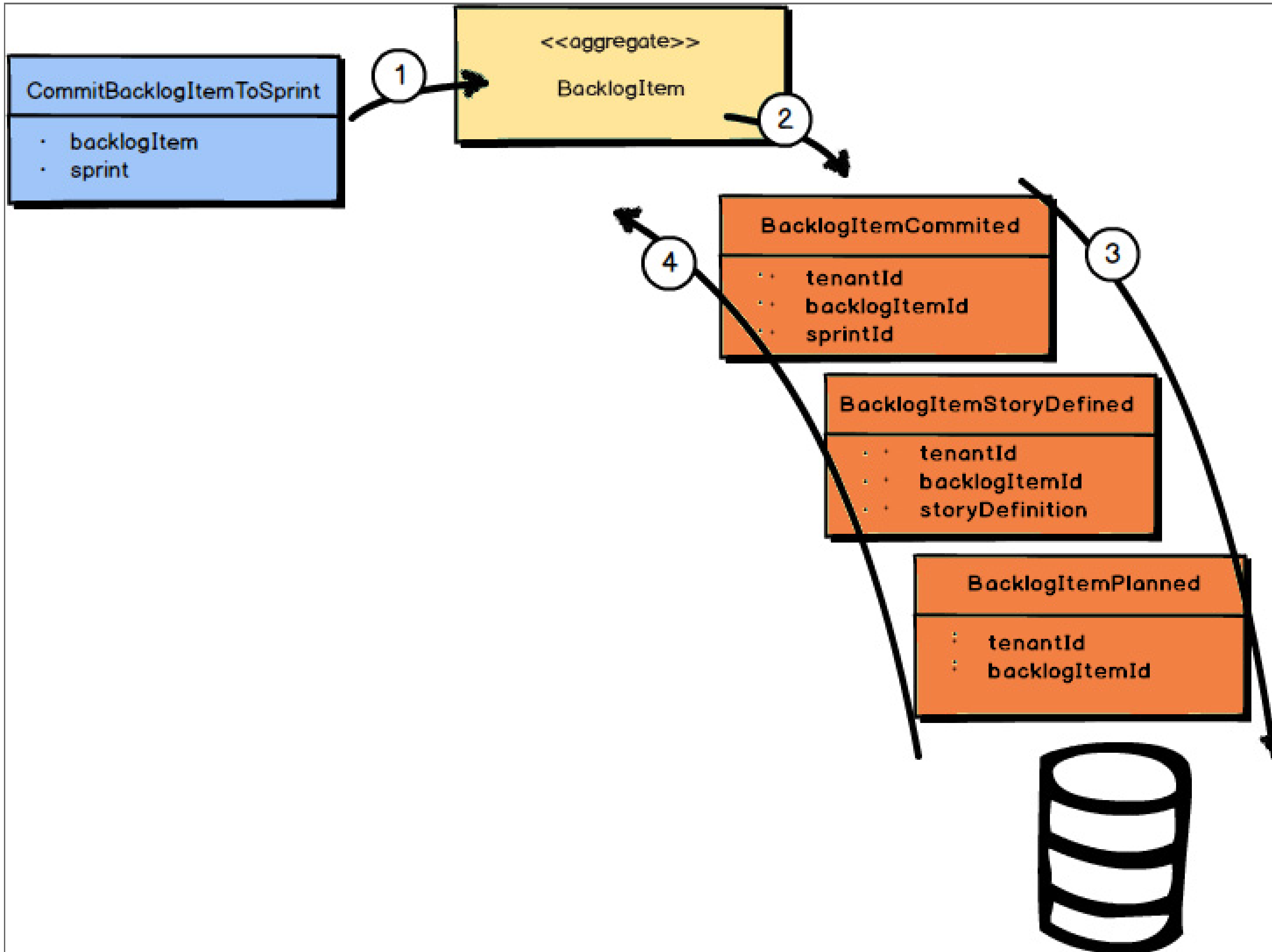


Domain Events Are Facts

- May be caused by a non-command source
- Example: time-based
- End of day/week/year
- Fiscal year ended
- Markets closed

FiscalYearEnded
Subscribing Bounded Context

Command



Command

- can be rejected by availability or by some kind of business validation
- need to generate one or more commands to carry a set of actions

Event Sourcing

- Can be described as persisting all Domain Events that have occurred for an Aggregate instance as a record of what changed about that Aggregate instance
- Rather than persisting the Aggregate state as a whole, you store all the individual Domain Events that have happened to it.

Stream Id	Stream Version	Event Type	Event Content
backlogItem123	1	BacklogItemPlanned	{ ... }
backlogItem123	2	BacklogItemStoryDefined	{ ... }
backlogItem123	3	BacklogItemCommitted	{ ... }
...	N	...	{ ... }
...	N	...	{ ... }
...	N	...	{ ... }

Projection

A projection refers to a read-only, denormalized view of the data that is derived from the events stored in the event store.
We can implement it as a view by aggregating all events (processing event stream from start to end). This view can is usually cached or precomputed.
There can be multiple projections based on event stream.

PERFORMANCE CONSCIOUS

- highest-performing Aggregates will be those that are cached in memory
- Using the Actor model with actors as Aggregates is one of the easier ways to keep your Aggregates' state cached.
- Using snapshot, the load time of your Aggregates, that have been evicted from memory, can be reconstituted optimally without reloading every Domain Event from an event stream

Question: how would you implement snapshot?

Modeling within Time Constraints

- Our goal is to learn about a business domain and refine a model
- Knowledge crunching takes time
- If we don't learn and model quickly, we will seem to fail even if we deliver
- We must timebox our modeling efforts
- We must not try to eliminate design

Questions?

That's it for today.

Speaker notes