
DOMAIN-DRIVEN
DESIGN
DISTILLED
VAUGHN VERNON

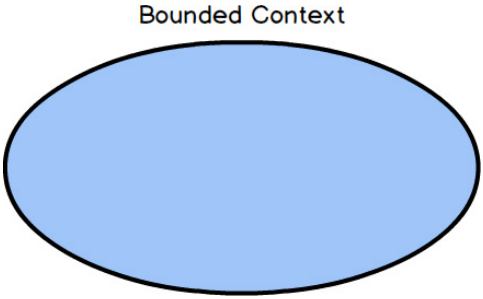
This quick reference book has been authored by Frederic Monjo. It is not a replacement for the original book, buy and read it first.

Although most contents comes from the original book, this book is not intended to disclose protected material. If you want to save this numeric version or print it, you must buy the book first.

The authored page size is A5, so that you can print it using a book page layout on a professional printer (fold the printed pages).

Strategic Design with Bounded Contexts and the Ubiquitous Language

In short, **DDD is primarily about modeling a Ubiquitous Language in an explicitly Bounded Context**.



A « Bounded Context » is a semantic contextual boundary. The components inside a Bounded Context are context specific and semantically motivated. You could think of it as part of your problem space.

Remember that a Bounded Context is where a model is implemented, and you will have separate software artifacts for each Bounded Context.

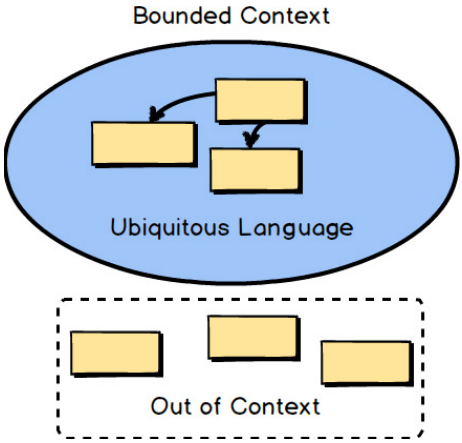
Within that Bounded Context, the language is called the « Ubiquitous Language » because it is both spoken among the team members and implemented in the software model.

When the Bounded Context is being developed as a key strategic initiative of your organization, it's called the Core Domain. A Core Domain is developed to distinguish your organization competitively from all others.

When someone on the team uses expressions from the Ubiquitous Language, everyone on the team understands what is meant with precision and constraints.

There should be one team assigned to work on one Bounded Context. There should also be a separate source code repository for each Bounded Context.

Your team owns the source code and the database and defines the official interfaces through which your Bounded Context must be used. It's a benefit of using DDD.

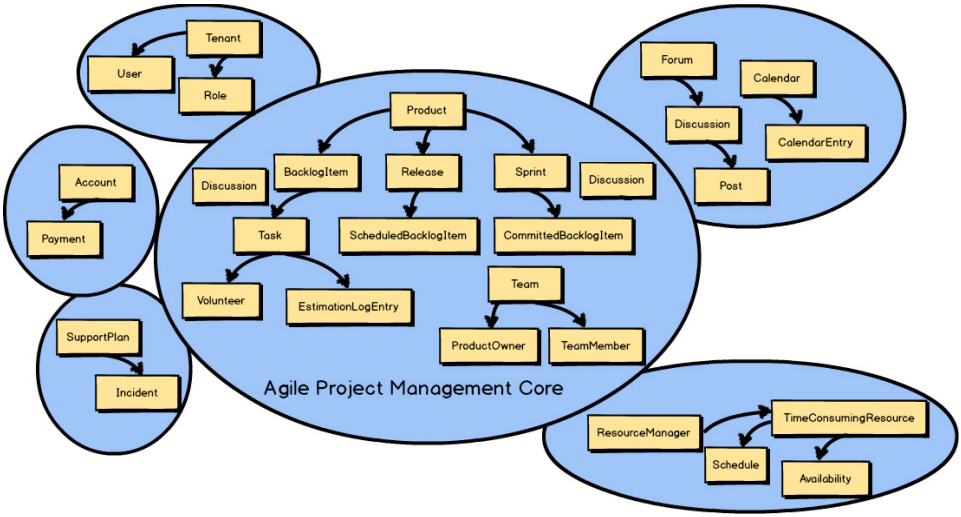


It's possible that people from other teams would have a different meaning for the same terminology, because their business knowledge is within a different context; they are developing a different Bounded Context. **Any components outside the context are not expected to adhere to the same definitions.**

Domain Experts and Business Drivers

The business's department or work group divisions can provide a good indication of where model boundaries should exist. You will tend to find at least one business expert per business function. Projects are organized according to business drivers and under an area of expertise.

Strategic Design with Subdomains



A Subdomain is a sub-part of your overall business domain. You can think of a Subdomain as representing a single, logical domain model. Subdomains can be used to logically break up your whole business domain so that you can understand your problem space on a large, complex project.

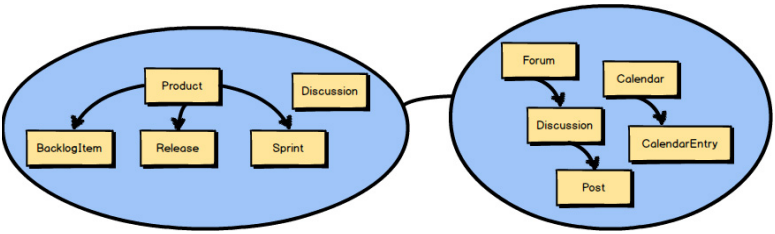
It is a clear area of expertise, assuming that it is responsible for providing a solution to a core area of your business. This implies that the particular Subdomain will have one or more Domain Experts who understand very well the aspects of the business that a specific Subdomain facilitates.

Three primary types of Subdomains :

- **Core Domain: This is where you are making a strategic investment** in a single, well-defined domain model.
- **Supporting Subdomain**: This is a modeling situation that calls for custom development, because an off-the-shelf solution doesn't exist.
- **Generic Subdomain**: This kind of solution may be available for purchase off the shelf.

When using DDD, a Bounded Context should align one-to-one (1:1) with a single Subdomain. This will keep your Bounded Contexts clean and focused on the core strategic initiative.

Strategic Design with Context Mapping



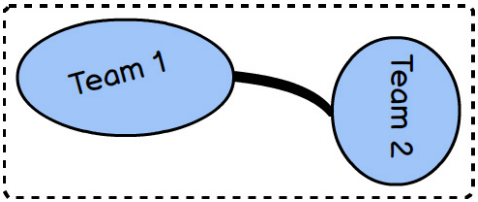
Core Domain would have to integrate with other Bounded Contexts. **That integration is known as Context Mapping.**

The line on the diagram indicates that the two Bounded Contexts are mapped in some way. There will be some inter-team dynamic between the two Bounded Contexts as well as some integration.

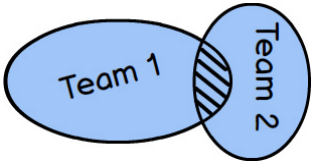
Considering that in two different Bounded Contexts there are two Ubiquitous Languages, this line **represents the translation that exists between the two languages**.

When we talk about Context Mapping, **what is of interest to us is what kind of inter-team relationship and integration is represented by the line** between any two Bounded Contexts.

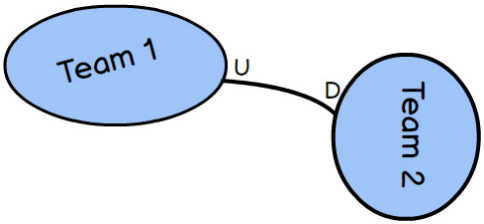
Kinds of mappings



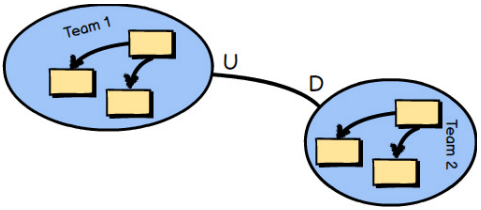
Create a Partnership to align the two teams with a dependent set of goals. It is said that **the two teams will succeed or fail together**. Since they are so closely aligned, they will meet frequently to synchronize schedules and dependent work, and they will have to use continuous integration to keep their integrations in harmony. **It can be challenging to maintain a Partnership over the long term**.



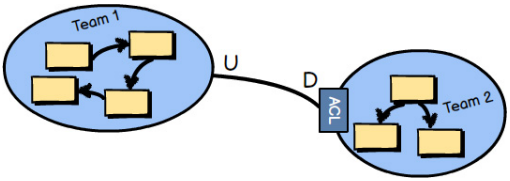
A Shared Kernel describes the relationship between two (or more) teams that share a small but common model. The teams must agree on what model elements they are to share. A Shared Kernel is **often very difficult to conceive in the first place, and difficult to maintain**, because you must have open communication between teams and constant agreement on what constitutes the model to be shared.



A **Customer-Supplier** describes a relationship where the Supplier is upstream (the U in the diagram) and the Customer is downstream (the D in the diagram). It's up to the Customer to plan with the Supplier to meet various expectations, but **in the end the Supplier determines what the Customer will get and when.**

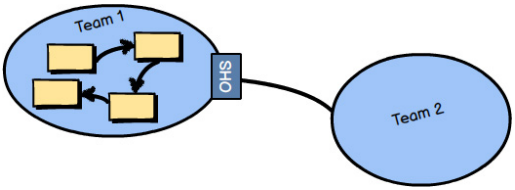


A Conformist relationship exists when the downstream team conforms to the upstream model as is. The upstream team has no motivation to support the specific needs of the downstream team.

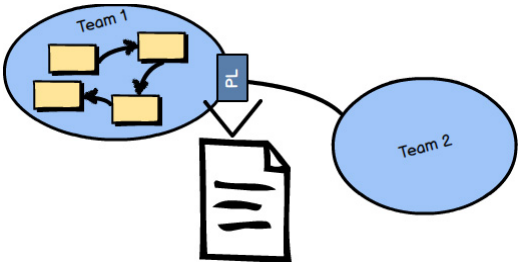


An **Anticorruption Layer is the most defensive Context Mapping relationship, where the downstream team creates a translation layer between its Ubiquitous Language and the Ubiquitous Language that is upstream to it**. The layer isolates the downstream model from the upstream model and translates between the two.

Whenever possible, you should try to create an Anticorruption Layer between your downstream model and an upstream integration model, so that you can produce model concepts on your side of the integration that specifically fit your business needs and that keep you completely isolated from foreign concepts. But the cost could be too high in various ways for some cases.



An Open Host Service defines a protocol or interface that gives access to your Bounded Context as a set of services. The protocol is “open” so that all who need to integrate with your Bounded Context can use it with relative ease.



A Published Language is a well-documented information exchange language enabling simple consumption and translation by any number of consuming Bounded Contexts. A Published Language can be defined with XML Schema, JSON Schema, or a more optimal wire format.

Often an Open Host Service serves and consumes a Published Language, which provides the best integration experience for third parties. This combination makes the translations between two Ubiquitous Languages very convenient.

In the least favorable of situations you may be forced to use database or file system integration, but let's hope that doesn't happen. **Database integration really should be avoided**, and if you are forced to integrate that way, you really should be sure to isolate your consuming model by means of an Anticorruption Layer.

RPC with SOAP

The idea behind RPC with SOAP is to **make using services from another system look like a simple, local procedure or method invocation**.

This carries the potential for complete network failure, or at least latency that is unanticipated when first implementing the integration. The main problem with RPC, using SOAP or another approach, is **that it can lack robustness**.

It would be in your best interests if there is a well-designed API that provides an Open Host Service with a Published Language.

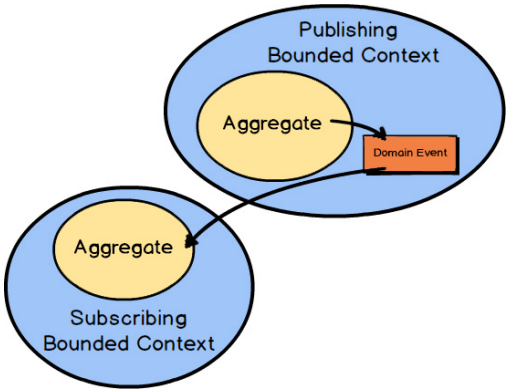
Restful HTTP

A service Bounded Context that sports a REST interface should provide an Open Host Service and a Published Language. **Resources deserve to be defined as a Published Language, and combined with your REST URIs they will form a natural Open Host Service**.

A common mistake made when using REST is to design resources that directly reflect the Aggregates in the domain model. Doing this forces every client into a Conformist relationship. Instead, resources should be designed synthetically to follow client-driven use cases.

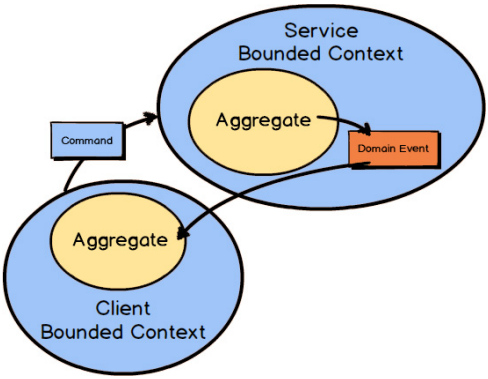
Messaging

Using messaging is **one of the most robust forms of integration because you remove much of the temporal coupling**. Since you already anticipate the latency of message exchange, you tend to build more robust systems because you never expect immediate results.

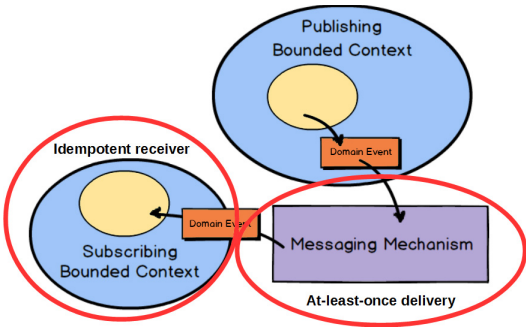


Typically an Aggregate in one Bounded Context publishes a Domain Event, which could be consumed by any number of interested parties. When a subscribing Bounded Context receives the Domain Event, some action will be taken based on its type and value.

A subscribing Bounded Context can always benefit from unsolicited happenings in the publishing Bounded Context.



Sometimes, a client Bounded Context will need to proactively send a Command Message to a service Bounded Context to force some action. **In such cases the client Bounded Context will still receive any outcome as a published Domain Event.**



The messaging mechanism should support At-Least-Once Delivery to ensure that all messages will be received eventually.

This also means that **the subscribing Bounded Context must be implemented as an Idempotent Receiver**. Whenever a message could be delivered more than once, the receiver should be designed to deal correctly with this situation.

Idempotent Receiver describes how the receiver of a request performs an operation in such a way that it produces the same result even if it is performed multiple times.

Due to the fact that messaging mechanisms always introduce asynchronous Request-Response communications, some amount of latency is both common and expected. Requests for service should (almost) never block until the service is fulfilled. Thus, **designing with messaging in mind means that you will always plan for at least some latency, which will make your overall solution much more robust from the outset**.

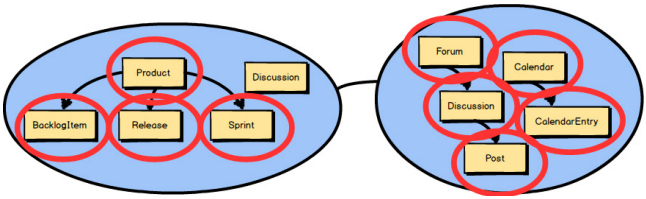
Enrichment versus Query-Back Trade-offs

Sometimes there is an advantage to enriching Domain Events with enough data to satisfy the needs of all consumers. Sometimes there is an advantage to keeping Domain Events thin and allowing for querying back when consumers need more data.

Enrichment, allows for greater autonomy of dependent consumers. On the other hand, there may be too much enrichment, and it may be a poor security choice to greatly enrich Domain Events.

Sometimes circumstances will call for a balanced blend of both approaches.

Tactical Design with Aggregates

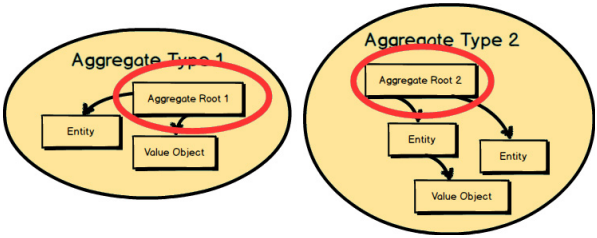


The concepts that live inside a Bounded Context are likely the Aggregates in your model.

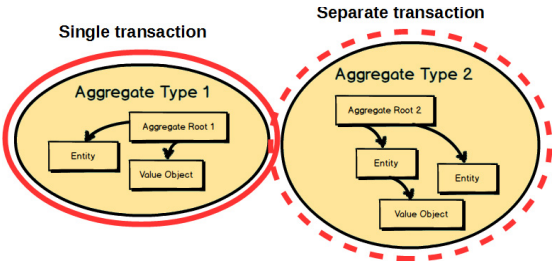
Each Aggregate is composed of one or more **Entities**, where one Entity is called the **Aggregate Root**. Aggregates may also have **Value Objects** composed on them.

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 times, an Entity will be mutable. **The main thing that separates an Entity from other modeling tools is its uniqueness—its individuality**.

A Value Object, or simply a Value, models an immutable conceptual whole. Within the model the Value is just that, a value. Unlike an Entity, it does not have a unique identity, and **equivalence is determined by comparing the attributes encapsulated by the Value type**.

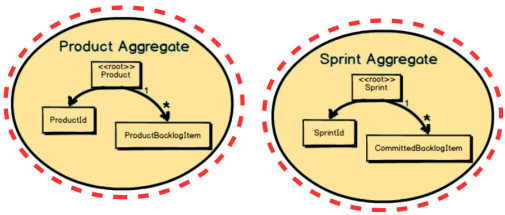


The Root Entity of each Aggregate owns all the other elements clustered inside it. The name of the Root Entity is the Aggregate's conceptual name. Choose a name that properly describes the conceptual whole that the Aggregate models.



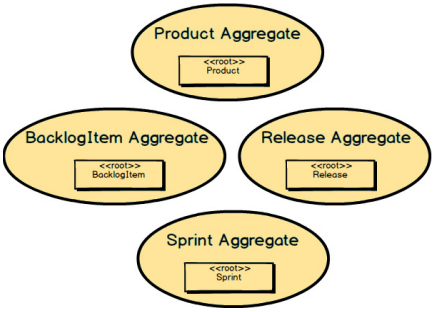
Each Aggregate forms a transactional consistency boundary. This means that within a single Aggregate, all composed parts must be consistent, according to business rules, when the controlling transaction is committed. The reasons for the transactional boundary are business motivated, because it is the business that determines what a valid state of the cluster should be at any given time.

Aggregates rules of thumb



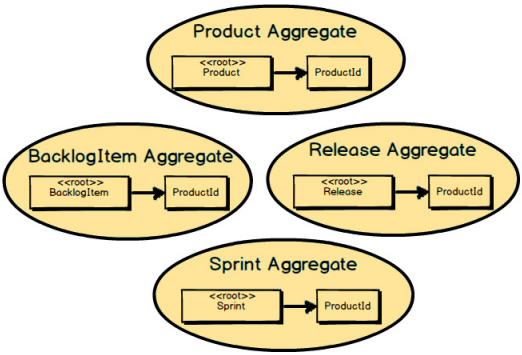
Rule 1: Protect Business Invariants inside Aggregate Boundaries

The business should ultimately determine Aggregate compositions based on what must be consistent when a transaction is committed. **At the end of a transaction this very specific business invariant must be met. The business requires it**.



Rule 2: Design Small Aggregates

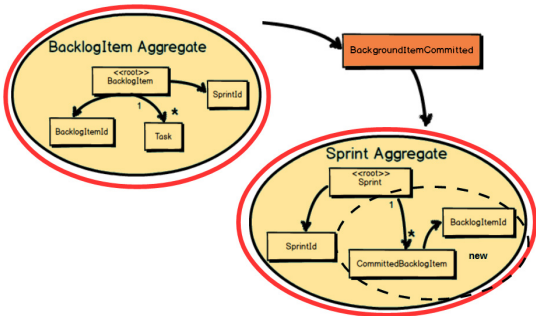
The memory footprint and transactional scope of each Aggregate should be relatively small. They will load quickly, take less memory, and are faster to garbage collect. More important, **they will have transactional success much more frequently**. This has the added benefit that each Aggregate will be easier to work on.



Rule 3: Reference Other Aggregates by Identity Only

This helps keep Aggregates small and **prevents reaching out to modify multiple Aggregates in the same transaction**.

Another benefit to using reference by identity only is that your Aggregates can be easily stored in just about any kind of persistence mechanism.



Rule 4: Update Other Aggregates Using Eventual Consistency

Domain Events are published by an Aggregate and subscribed to by an interested Bounded Context. The interested Bounded Context can be the same one from which the Domain Event was published.

One big, nasty hook is the Anemic Domain Model: this is where you are using an object-oriented domain model, and all of your Aggregates have only public accessors (getters and setters) but no real business behavior. Also watch out for leaking business logic into the Application Services above your domain model.

Place your business logic in your domain model

The first thing you must do is create a class for your Aggregate Root Entity. Every Aggregate Root Entity must have a globally unique identity.

Next you capture any intrinsic attributes or fields that are necessary for finding the Aggregate. Of course, you can add simple behavior such as read accessors (getters) for intrinsic attributes.

If you expose public setter methods, it could quickly lead to anemia, because the logic for setting values on Product would be implemented outside the model. Think hard before doing this, and keep this warning in mind.

Finally, you add in any complex behavior exposed in public functions.

Right-sizing Aggregates

Consider these design steps that will help you reach consistency boundary goals:

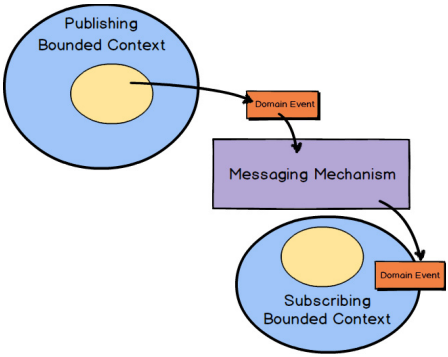
1. **Start by creating every Aggregate with just one Entity, which will serve as the Aggregate Root**. Don't even dare to place two Entities in a single boundary.
2. **Populate each of the Entities with the fields that you believe are most closely associated with the single Root Entity**. One big hint here is to define **every field that is required to identify and find the Aggregate**, as well as any **additional intrinsic fields that are required for the Aggregate to be constructed and left in a valid initial state**.
3. Look at each of your Aggregates one at a time, and **ask the Domain Experts if any other Aggregates you have defined must be updated in reaction to changes made to Aggregate A1**. Now ask the Domain Experts **how much time may elapse until each of the reaction-based updates may take place**. This will lead to two kinds of specifications: (a) immediately, or (b) within N seconds/minutes/hours/days. One possible way to find the correct business threshold is by presenting an exaggerated time frame (such as weeks or months) that is obviously unacceptable. This will likely cause business experts to respond with an acceptable time frame.
4. **For each of the immediate time frames (3a), you should strongly consider composing those two Entities within the same Aggregate boundary**. That means, for example, that Aggregate A1 and Aggregate A2 will actually be composed into a new Aggregate A[1,2]. Now Aggregates A1 and A2 as they were previously defined will no longer exist. There is only Aggregate A[1,2].
5. **For each of the reacting Aggregates that can be updated following a given elapsed time (3b), you will update these using eventual consistency**.

Be careful that the business doesn't insist that every Aggregate fall within the 3a specification (immediate consistency). It is very unlikely that the business really needs immediate consistency in every case. **Considering what the business would have to do if it ran its operations only by means of paper systems can provide some**

worthwhile insights into how various domain-driven operations should work within a software model of the business operations.

You should also design your Aggregates to be a sound encapsulation for unit testing. Complex Aggregates are hard to test. Following the previous design guidance will help you model testable Aggregates. What we are concerned with here is testing that the Aggregate correctly does what it is supposed to do. You want to push on all the operations to ensure the correctness, quality, and stability of your Aggregates.

Tactical Design with Domain Events

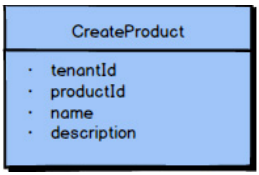


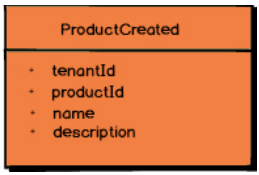
A Domain Event is a record of some business-significant occurrence in a Bounded Context. Domain Events are conceptualized and become a part of your Core Domain.

A business domain provides **causal consistency if its operations** that are causally related—one operation causes another—**are seen by every dependent node of a distributed system in the same order**. Causally related operations must occur in a specific order, and thus one thing cannot happen unless another thing happens before it.

This sort of causal, linearized system architecture can be readily achieved through the creation and publication of correctly ordered Domain Events. Domain Events become a reality in your domain model and can as a result be published and consumed in your own Bounded Context and by others. It's a very powerful way to inform interested listeners of important occurrences that have taken place.

You must show care in how you name your Domain Event types. **The words you use should reflect your model's Ubiquitous Language**. These words will form a bridge between the happenings in your model and the outside world. **It's vital that you communicate your happenings well**.



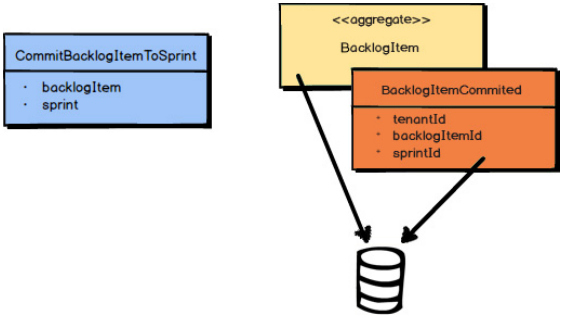


Your Domain Event type names should be a statement of a past occurrence, that is, a verb in the past tense. The Domain Event should **hold all the properties that were provided with the command that caused it to be created.**

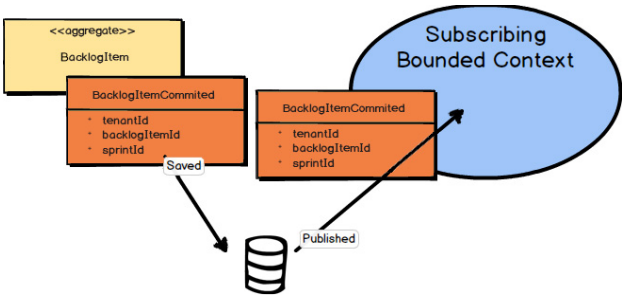
There are times when a Domain Event can be enriched with additional data. This can be especially helpful to consumers that don't want to query back on your Bounded Context to obtain additional data that they need. Even so, **you must be careful not to fill up a Domain Event with so much data that it loses its meaning**.

Also, **do not include the full state of the related entity**, this would make the event very unclear, because the consumer would have to compare the latest event to the previous one in order to understand what actually occurred to the Aggregate.

Ensuring causality relationships



It's important that the modified Aggregate and the Domain Event be saved together in the same transaction. If you are using an object-relational mapping tool, you would save the Aggregate to one table and the Domain Event to an event store table, and then commit the transaction.

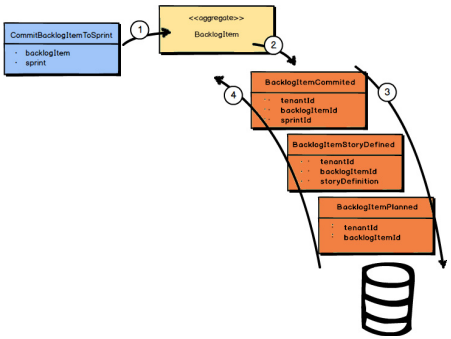


If you are using Event Sourcing, the state of the Aggregate is fully represented by the Domain Events themselves. **Just saving the Domain Event in its causal order doesn't guarantee that it will arrive at other distributed nodes in the same order**.

It is also the responsibility of the consuming Bounded Context to recognize proper causality. It might be the Domain Event type itself that can indicate causality, or **it may be metadata associated with the Domain Event, such as a sequence or causal identifier**.

Although often it is a user-based command emitted by the user interface that causes an event to occur, sometimes Domain Events can be caused by a different source. This might be from a timer that expires. You can't reject the fact that some time frame has expired, and if the business cares about this fact, the time expiration is modeled as a Domain Event, and not as a command.

A command is different from a Domain Event in that a command can be rejected as inappropriate in some cases. A Domain Event is a matter of history and cannot logically be denied. In response to a time-based Domain Event it could be that the application will need to generate one or more commands in order to ask the application to carry out some 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.

Reapplying the event stream to the Aggregate allows its state to be reconstituted from persistence back into memory.

The event store is just a sequential, append-only storage collection where all Domain Events are appended. The mechanism is extremely fast, so you can plan to **have very high throughput, low latency, and be capable of high scalability**.

Event Sourcing saves a record of everything that has ever happened in your Core Domain, at the individual occurrence level. This can be very helpful to your business for many reasons that you won't realize until later. There are also technical advantages: software developers can use event streams to examine usage trends and to debug their source code.

When you use Event Sourcing you are almost certainly obligated to use CQRS.

Acceleration tools

When using DDD we are on a quest for deep learning about how the business works, and then to model software based on the extent of our learning. It's really a process of learning, experimenting, challenging, learning more, and modeling again. **We need to crunch and distill knowledge in great quantities and produce a design that is effective in meeting the strategic needs of an organization. The challenge is that we have to learn quickly.**

If we don't deliver on time and within budget, no matter what we have achieved with the software, we seem to have failed. Unfortunately, **one common response to this negative pressure is to try to economize and shorten timelines by eliminating design**.

Event Storming

Event Storming is a rapid design technique that is meant to engage both Domain Experts and developers in a fast-paced learning process. It is focused on the business and business process rather than on nouns and data. It is an approach to rapid learning and software design that got everybody in the room very directly involved in the process.

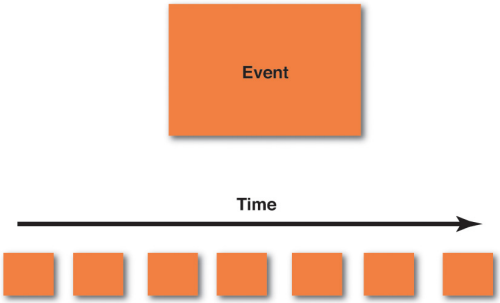
Advantages

- a very tactile and visual approach. Everyone gets a pad of sticky notes and a pen and is responsible for contributing to the learning and design sessions.
- both business people and developers stand on equal ground as they learn together.
- focuses everyone on events and the business process rather than on classes and the database.
- fast and cheap to perform. If you write something on a sticky note that you later decide doesn't work, you wad up the sticky note and throw it away. It costs you only a penny or two for that mistake, and no one is going to resist the opportunity to refine due to effort already invested.
- your team will have breakthroughs in understanding. Period. It happens every time.
- everybody learns something. Storming out a model helps everyone clear up misunderstandings and move forward with a unified direction and purpose.
- you are also identifying problems in both the model and in understanding as early and quickly as possible. You can use Event Storming for both big-picture and design-level modeling.

List of people and supplies

- having the right people is essential, which means the Domain Expert(s) and developers who are to work on the model
- everyone should come with an open mind that is free of strict judgment. There is time to refine later, and refinement is fast and cheap.
- an assortment of colors of sticky notes, and plenty of them. You don't want your sticky notes falling on the floor.
- one black marker pen for each person, which will enable handwriting to show up bold and clear. Fine-tip markers are best.
- find a wide wall where you can model. Width is more important than height.
- a long roll of paper. Hang the paper on the wall using strong tape. Some may decide to work on whiteboards. This may work for a while, but the sticky notes tend to lose adhesion over time on whiteboards.

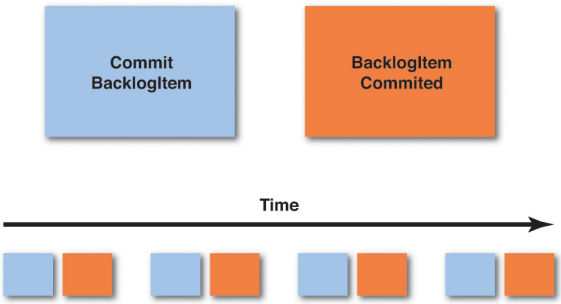
List of steps



1. Storm out the business process by **creating a series of Domain Events** on sticky notes. The most popular color to use for Domain Events is orange, it makes the Domain Events stand out most prominently on the modeling surface.

   Basic guidelines:

   - We have **our first and primary focus on the business process, not on the data and its structure**.
   - The name of each Domain Event should be **a verb stated in the past tense**.
   - Place the sticky notes **in time order horizontally**
   - A Domain Event that happens in parallel with another according to your business process can be located under the Domain Event that happens at the same time. **Use vertical space to represent parallel processing.**
   - **Clearly mark trouble spots in your business process**, with a purple/red sticky note and some text that explains why it's a problem.
   - Sometimes the outcome of a Domain Event is **a Process that needs to run**. Place it on a lilac sticky note, and draw an arrow line from the Domain Event to the named Process.

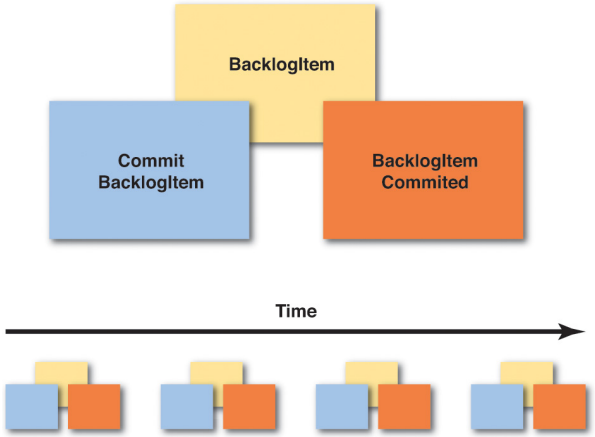


2. **Create the Commands that cause each Domain Event. The Command should be stated in the imperative**, such as « CreateProduct » and « CommitBacklogItem ».

   Often a Command will be the outcome of some user gesture, and that Command, when carried out, will cause a Domain Event. Sometimes a Domain Event will be the outcome of a happening in another system.

   Basic guidelines:

   - Use **light blue sticky notes** for Commands
   - Place the light blue sticky note of the **Command just to the left of the Domain Event that it causes**. They are associated in pairs.
   - If there is **a specific user role** that performs an action, and it is important to specify, you can place a **small, bright yellow sticky note on the lower left corner of the light blue Command** with a stick figure and the name of the role.
   - Each Command that causes a Process to be executed should be captured and named on a lilac sticky note. Draw **an arrow line from the Command to the named Process**. The Process will actually cause one or more Commands and subsequent Domain Events, and if you know what those are now, create sticky notes for them and show them emitting from the Process.
   - It is possible that creating Commands will cause you to think about Domain Events that you didn't previously envision. Go ahead and address this discovery by placing the newly discovered Domain Event on the modeling surface.
   - You may also find that there is only one Command that causes multiple Domain Events. That's fine; model the one Command and place it to the left of the multiple Domain Events that it causes.

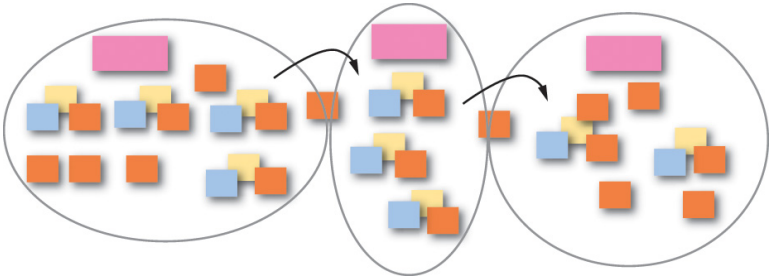


3. **Associate the Entity/Aggregate** on which the Command is executed and that produces the Domain Event outcome.

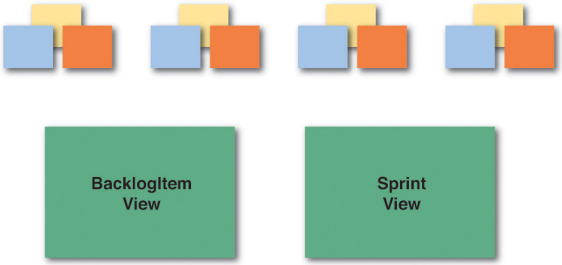
   Basic guidelines:

   - If the business people don't like the word Aggregate, or if it confuses them in any way, you should use another name. Usually they can understand Entity, or you could just call it Data.
   - The Command and Domain Event pairs should adhere to the lower part of the Aggregate sticky to indicate that they are associated.
   - **Create the same Aggregate noun on multiple sticky notes** and place them repeatedly on the timeline where the corresponding Command/Event pairs occur.
   - You may discover new Domain Events. Don't ignore these.
   - You may also discover that some of the Aggregates are too complex, and you need to break these into a managed Process (lilac sticky). Don't ignore these opportunities.

If you are using Event Sourcing, as described in the previous chapter, you have already come a long way toward understanding your Core Domain implementation, because there is a large overlap in Event Storming and Event Sourcing.



4. **Draw boundaries and lines with arrows to show flow** on your modeling surface.
    - You will very likely find boundaries under the following conditions: departmental divisions, when different business people have conflicting definitions for the same term, or when a concept is important but not really part of the Core Domain.
    - Use your black marker pens to draw on the paper modeling surface. Show context and other boundaries. Use solid lines for Bounded Contexts and dashed lines for Subdomains.
    - Place the pink sticky notes inside various boundaries and put the name that applies inside the boundary on those sticky notes. This names your Bounded Contexts.
    - Draw lines with arrowheads to show the direction of Domain Events flowing between Bounded Contexts.



5. **Identify the various views that your users will need** to carry out their actions, and important roles for various users.
    - If you decide to show any views, they should be those that are significant and require some special care in creating. These view artifacts can be represented by green sticky notes on the modeling surface. If it helps, draw a quick mockup.
    - Use bright yellow sticky notes to represent various important user roles. Again, show these only if you need to communicate something of significance about the user's interaction with the system, or something that the system does for a specific role of user.

Other tools

Remember, this is about learning and communicating a design. Use whatever tools you need to model as a close-knit team. Just be careful to reject ceremony, because that's going to cost a lot.

- Introduce **high-level executable specifications** that follow the given/when/then approach (Behavior Driven Development). These are also known as acceptance tests.
- Try **Impact Mapping** to make sure the software you are designing is a Core Domain and not some less important model.
- Look into **User Story Mapping** by Jeff Patton.
- You will have to "buy" knowledge about your product, and sometimes the payment is a spike (technical investigation).

Identifying Tasks and Estimating Effort

Each of the Domain Events, Commands, and Aggregates that you storm out in your paper model can be used as estimation units. Create a table like this:

Component Type	Easy (Hours)	Moderate (Hours)	Complex (Hours)
Domain Event	0.1	0.2	0.3
Command	0.1	0.2	0.3
Aggregate	1	2	4
. . .	. . .	. . .	. . .

Here is how the table works:

1. Create one column for Component Type and three other columns, one each for Easy, Moderate, and Complex.
2. Create one row for each component type in your architecture. Shown are Domain Event, Command, and Aggregate types. However, don't limit yourself to those. Create a row for the various user interface components, services, persistence, Domain Event serializers and deserializers, and so on.
3. Fill in the hours or fraction of an hour needed for each level of complexity: easy, moderate, and complex.
4. When you know the backlog item tasks that you will work on, obtain a metric for each of the tasks and identify it clearly.
5. Add up all the estimation.

As you execute each sprint, tune your metrics to reflect the hours or fractions of hours that were actually required.

Timeboxed Modeling

Concrete scenarios and Event Storming are two tools that should be used together:

1. Perform a quick session of Event Storming, perhaps just for an hour or so. You will almost certainly discover that you need to develop more concrete scenarios around some of your quick modeling discoveries.
2. Partner with a Domain Expert to discuss one or more concrete scenarios that need to be refined.
3. Create a set of acceptance tests (or executable specifications) that exercise each of the scenarios.
4. Create the components to allow the tests/specifications to execute. Iterate (briefly and quickly) as you refine the tests/specifications and the components until they do what your Domain Expert expects.

Interacting with Domain Experts

One of the major challenges of employing DDD is getting time with Domain Experts, and without overdoing it. Unless you make modeling sessions fun and efficient, you stand a good chance of losing their help at just the wrong time.

So the first questions to answer are “When do we need time with Domain Experts? What tasks do they need to help us perform?

- Always include them in Event Storming activities.
- You will need their input on discussions and the creation of model scenarios.
- To review tests to verify model correctness.
- To refine the Ubiquitous Language and its Aggregate names, Commands, and Domain Events, which are determined by the entire team. Ambiguities are resolved through review, questions, and discussion.

How much time should you require of them for each of these responsibilities?

- Event Storming sessions should be limited to a few hours (two or three) each.
- Generous amounts of time for scenario discussion and refinement.
- Discuss and iterate on one scenario over perhaps 10 to 20 minutes of time
- Some time to review what you have written
- One test every one to two minutes, or thereabouts