Domain-driven development allows us to plan a microservice architecture by decomposing the larger system into self-contained units.

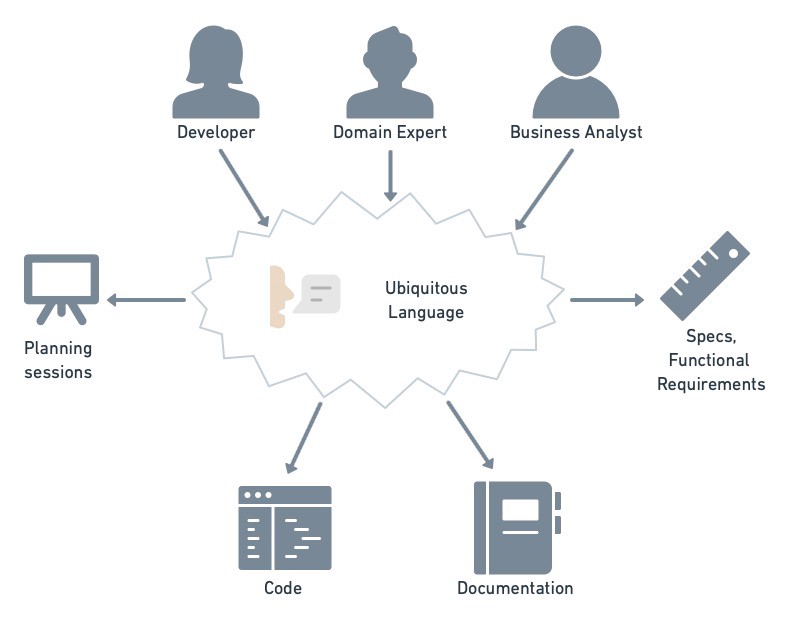
* Understanding responsibilities
* Identifying relationship

Domain-driven design

It is doftware design method where in the developers construct models to understand business requirements of domain.

Conceptual foundation for developing software.

Developers are smart by they cant be specialist in all fields.They need to collaborate with domain experts to guarantee the code is aligned with **business rules** and **client needs**



Developers and domain experts use unified language to share knowledge ,document ,plan and code .

The two most important DDD concepts for microservice architecture are : bounded contexts and context maps.

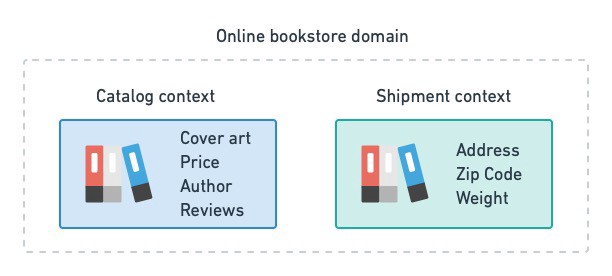
Bounded Context

* A bounded context BC is the **space in which a term has definite and unambiguous** meaning .( book has 2 meaning – witten piece of work or it may be to reserve a room)

Before DDD , it was common practice to attempt to find model that spanned complete domain.

Larger the domain ,more difficult to find consistent and unified model.

DDD’s solution is toidentify BC so that domain can be broken down into manageable sub-domains.



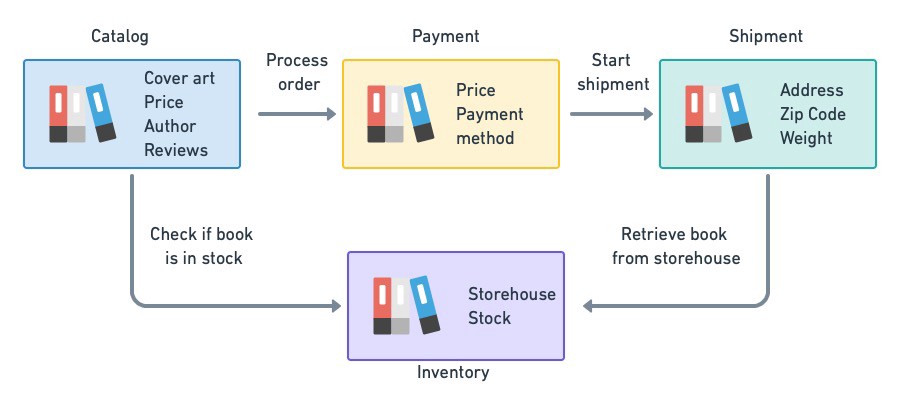
In software , we need to be exact.thats why defining BC is critical.It gives precise vocabulary called ubiquitous language.(language between developers and domain experts)

This language is present through design process, project documentation and code.

Context Map:

Presence of BC anticipates the need of communication channels.

In DDD , these relationship are depicted in form of context map.



DDD for microservices:

It takes place in 2 phases :

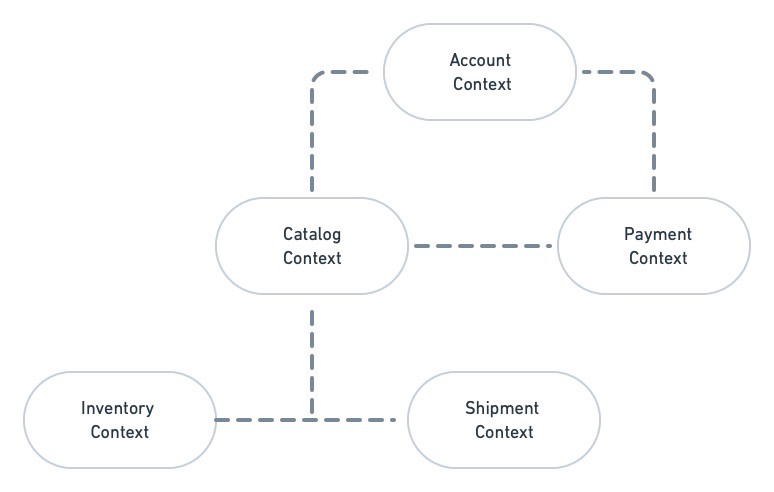
1.Strategic Phase : WE identify BC’s and map them out in a context map.

2. Tactical phase we model each BC according to business rules of sub domain.

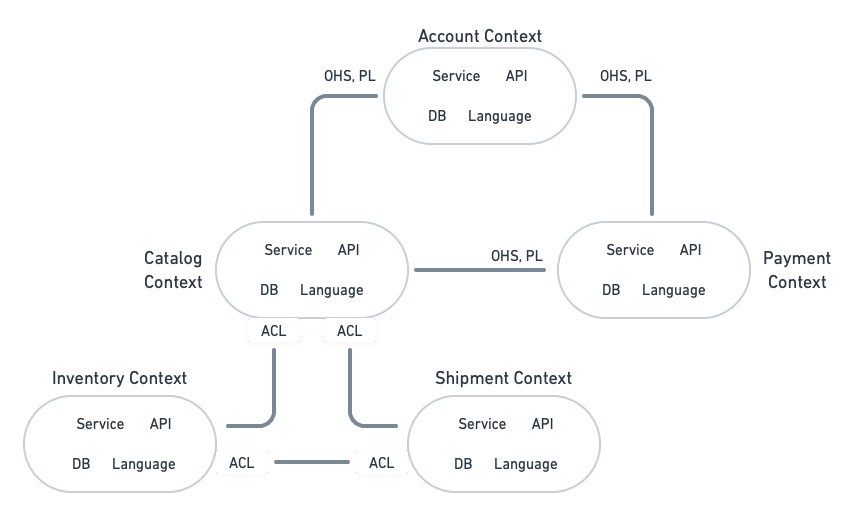
Strategic Phase :

* Developer
* Domain Experts
* Product owners
* Business analysts

For brain storming and initial plan.With aid of facilitator this can take the form of Event stroming workshop session.

* Build models and identify business requirements starting from significant events in the domain.
* In strategic DDD, we take high-level , top-to-bottom approach to design .
* Analyzing the domain in order to determine business rules and we derive BC’s.
* 
* Every BC represents an opportunity to implement at least one microservice.
* 
* How BC will communicate.
* There are 7 types of relationship . Three are (Shared Kerner , customer /supplier ,conformist) which imply tight coupling and do not want in microservices.
* Other 4 types are
* Open Host Service:
  + Service provider defines open protocol for others to consume .
  + Open-ended relationship
  + It is up to the consumer to conform on the protocol
* Published Language
  + This type of communication can be combined with well known language (XML,JSON,GraphQL) which fits the domain
* Anti corruption layer :
  + It is abstraction and translation wrapping layer implemented in front of downstream
  + When something changes upstream the consumer service only needs to update the ACL
* Separate Ways :
  + This happens when integration between 2 services is found .
  + Opposite of relationship
  + BS’s have no connection and do not need to interact.

At end of strategic analysis ,we get context map dealing the BC’s and their relationship



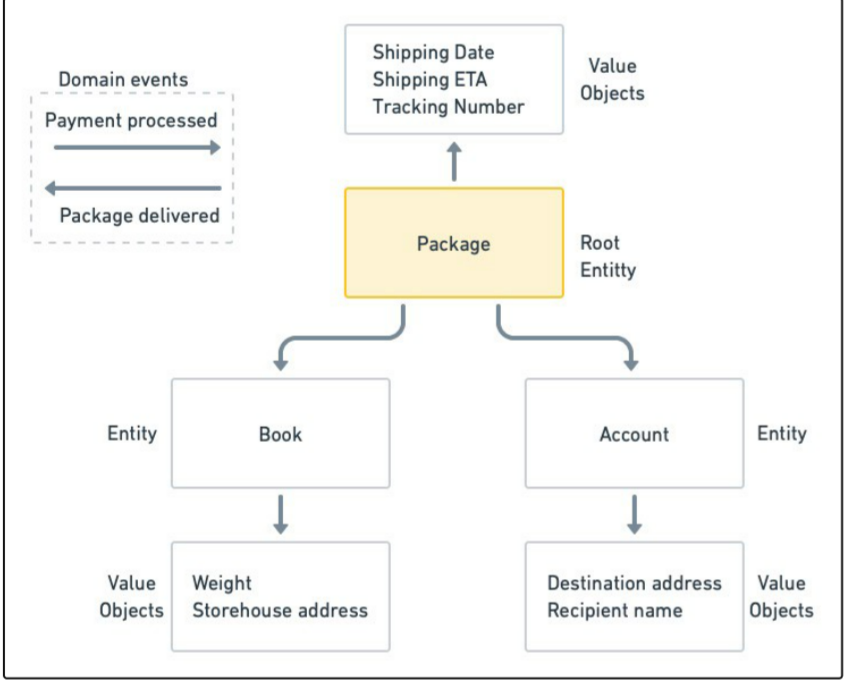
Tactical Phase :

Main building blocks of our model are :

* Entities : Entities are the objects with an identity that persist over time .

Value Objects : They are immutable values without identity.they represent primitives of model ,dates ,times ,coordinates or currencies.

* **Aggregates**: aggregates create relationships between entities and value objects. They represent a group of objects that can be treated as a single unit and are always in a consistent state. For example, customers place orders and own books, so the entities customer, order, and book can be treated as an aggregate. Aggregates must always be referenced by a main entity, called the *root entity*.
* **Domain services**: these are stateless services that implement a piece of business logic or functionality. A domain service can span multiple entities.
* **Domain events**: essential for microservice design, domain events notify other services when something happens. For instance, when a customer buys a book, a payment is rejected, or that a user has logged in. Microservices can simultaneously produce and consume events from the network.
* **Repositories**: repositories are persistent containers for aggregates, typically taking the form of a database.
* **Factories**: factories are responsible for creating new aggregates.



# Domain-Driven Design is iterative

While it may appear that we must first write an exhaustive description of the domain before we can begin working on the code, the reality is that DDD, like all software design, is an iterative process.

On paper, bounded contexts and context maps may appear OK, but when implemented, they may translate into services that are too big to be rightly called microservices. Conversely, chatty microservices with overlapping responsibilities may need to be merged into one.

As development progresses and you have a better understanding of the domain, you’ll be able to make better judgments, enhance models, and communicate more effectively.

# More ways of designing microservices

DDD is undoubtedly a theory-heavy design pattern. As a result, it is only recommended when the system under development is complex enough to warrant the extra planning work.

Other methods such as [Test-Driven Development](https://semaphoreci.com/blog/test-driven-development) (TDD) or [Behavior-Driven Development](https://semaphoreci.com/community/tutorials/behavior-driven-development" \t "_blank) (BDD) may be enough for smaller, simpler systems. TDD is the fastest to start with and works best when working on single microservices or even with applications consisting of only a few services.

On a bigger scale, we can use BDD, which forces us to validate the wholesale behavior with integration and acceptance tests. BDD may work well if you work on low to medium-complexity designs, but once you hit a certain threshold, maintaining the tests can slow you down.

You can also combine these three patterns, choosing the best one for each stage of development. For example:

1. Identify microservices and their relationships with strategic DDD.
2. Model each microservice with tactical DDD.
3. Since each team is autonomous, they can choose to adopt BDD or TDD (or a mix of both) for developing a microservice or a cluster of microservices.

# Learn more about Domain-Driven Design

DDD can feel dauting to learn and implement, but its value for developing a microservice architecture is well worth the effort. If you found the information in this article interesting, I recommend picking up the relevant books by [Eric Evans](https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215) and [Vaughn Vernon](https://kalele.io/books/) to learn more.

Originally published at [https://semaphoreci.com](https://semaphoreci.com/blog/domain-driven-design-microservices) on June 21, 2022.