**CORE DOMAIN**

* IS THE MOST IMPORTANT PART OF THE DDD.
* ITS THE PART OF THE BUSINESS THAT DIFFERENTIATES OUR ORGANIZATION OR PROJECT FROM OTHERS
* ITS THE PART OF THE BUSINESS ON WHICH OUR ORGANIZATION MUST EXCEL

**STRATEGIC DECING**

**BOUNDED CONTEXT AND UBIQUITOUS LANGUAGE**

* **Bounded Context:**
  + Is a boundary, where each of its components have a specific meaning and do, a specific thing.
* **Ubiquitous Language**
  + It’s a rigorous, strict, exact, stringent, and tight language that is spoken and understood by all the members of the team
  + Its semantics are specific for each Bounded context of which they make part.

**BIG BALL OF MUD**

* Is when a highly coupled system is developed.
* We cannot distinguish between the core domain and its sub domains.
* The Ubiquitous Language is fractured.

**How to challenge your assumptions and unify mental models**

* Write down the steps needed for a specific functionality of the core domain, with all its constraints, help us to discover possible new restrictions, characteristics or dependencies that may have been overlooked.
* The write down scenario must not been writer as if humans use it, instead must be how the actual software is going to be used to give support to the components of the problem to be solved.
* We must use nouns, verbs and adverbs in the ubiquitous language that we are using to describe the scenario.

**Architectural components found inside a Bounded Context**

* The most important part is the ***CORE DOMAIN***
* All the components must be specific for the business logic of the CORE DOMAIN
* Its Ubiquitous Language must be semantically specific for the business logic that is been developed
* **ARCHITECTURE MAIN LAYERS:**
  + INPUT ADAPTER: an abstraction of an APPLICATION SERVICE that is been exposed to the environment to be consumed
  + APPLICATION SERVICE: where the user cases and transactions are made.
  + DOMAIN MODEL: where the business logic and domain events are executed.
  + OUTPUT ADAPTER: where persistence managers and message sender reside.

**Sub domains**

It’s a sub part of an overall business domain which represents a single logical model. In other words, its in charge of give solution to one of the many areas of expertise that the business requires.

**Types of subdomains**

* **CORE DOMAIN: The part of the business that distinguish your organization form other. Is where most of the resources are invested.**
* **Supporting subdomain:** It’s a modelling situation that calls for custom development, because the solution may not exist or the ones that can be acquired are not in line with our necessities. It can be outsourced to not confuse it with the **CORE DOMAIN** and invest unnecessary resources on it.
  + ***It is an important software model, because our CORE DOMAIN cannot function properly without it.***
* **Generic subdomain:** This kind of solution may be available for purchase off the shelf but may also be outsourced or even developed in house by a team that doesn’t have the kind of elite

Developers that you assign to your ***CORE DOMAIN*** or even a lesser *Supporting Subdomain.*

**Dealing with legacy systems**

* *Legacy systems may have several Logical* domain models that exist inside that one legacy system. Think of each of those logical domain models as a *Subdomain.*
* The legacy system seems less monolithic and muddy if we imagine separate Ubiquitous Languages, at least for the sake of understanding how we must integrate with it. Thinking about and discussing such legacy systems using Subdomains helps us cope with the harsh realities of a large entangled model.

**CONTEXT MAPPING**

Is a tool that allow us create relationships and dependencies among the bounded contexts of our project.

**Types of mappings:**

* **Partnership:** Each team is responsible for one Bounded Context. They create a Partnership to align the two teams with a dependent set of goals.
* **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. It’s possible that only one of the teams will maintain the code, build, and test for what is shared. A Shared Kernel is often very difficult to conceive.
* **Customer-Supplier:** describes a relationship between two Bounded Contexts and respective teams, where the Supplier is upstream (the U in the diagram) and the Customer is downstream (the D in the diagram). The Supplier holds sway in this relationship because it must provide what the Customer needs. 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.
* **Conformist:** itexists when there are upstream and downstream teams, and the upstream team has no motivation to support the specific needs of the downstream team. For various reasons the downstream team cannot sustain an effort to translate the Ubiquitous Language of the upstream model to fit its specific needs, so the team conforms to the upstream model as is. For example: When the upstream is a large scale supplier, like AWS.
* **Anticorruption layer:** is the most defensive Context Mapping relationship, where the downstream team creates a translation layer between its Ubiquitous Language (model) and the Ubiquitous Language (model) that is upstream to it. The layer isolates the downstream model from the upstream model and translates between the two. Thus, this is also an approach to integration. 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.
* **Open Host Services:** 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. The services offered by the application programming interface (API) are well documented and a pleasure to use.
* **Published Language:** is a well-documented information exchange language enabling simple consumption and translation by any number of consuming Bounded Contexts. Consumers who both read and write can translate from and into the shared language with confidence that their integrations are correct. Such a Published Language can be defined with XML Schema, JSON Schema, or a more optimal wire format, such as Protobuf or Avro**. OFTEN AN *OPEN HOST SERVICE* SERVES AND CONSUMES A PUBLISHED LANGUAGE.**
* **Separate Ways:** describes a situation where integration with one or more Bounded Contexts will not produce significant payoff through the consumption of various Ubiquitous Languages. Ergo, the relationship will be not made.

**Dealing with a BIG BALL OF MUD with context mapping**

If you must integrate with one or more, try to create an Anticorruption Layer against each legacy system in order to protect your own model from the cruft that would otherwise pollute your model with the incomprehensible morass. Whatever you do, don’t speak that language!

**TACTICAL DESIGN WITH AGGREGATES**