**What is Domain Driven Design?**

* It is a way of thinking and a set of priorities, aimed at accelerating software projects that have to deal with **complicated domains**.
* A **model driven** software design approach used to tackle the complexity of software projects.
* Collection of principles and patterns that help developers craft **elegant systems**.

**Advantages:**

* Loose coupling
* Flexibility
* Testability
* Maintenance

### Disadvantages

* Domain Experties
* Low interactions
* Development Costs

**Definitions:**

* **Domain**: a sphere of knowledge or activity. What an organization does and the world it does it in.
* **Model**: A system of abstractions that describes selected aspects of a domain and ignores extraneous detail. Explains a complex domain in a simple way.
* A **model** is a distilled form of **domain** knowledge, assumptions, rules and choices.

**DDD is a Set of Driving Principles:**

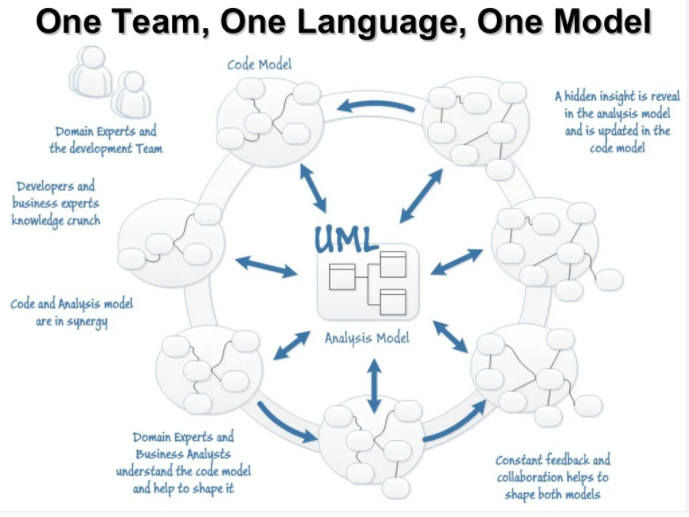
* Speak a **Ubiquitous Language (shared/common language)** within an explicitly Bounded Context.
* Explore **models** in a creative collaboration of domain practitioners and software practitioners.
* Focus on the **Core Domain.**
* Model and implementation are bound, developers are also responsible for the model.

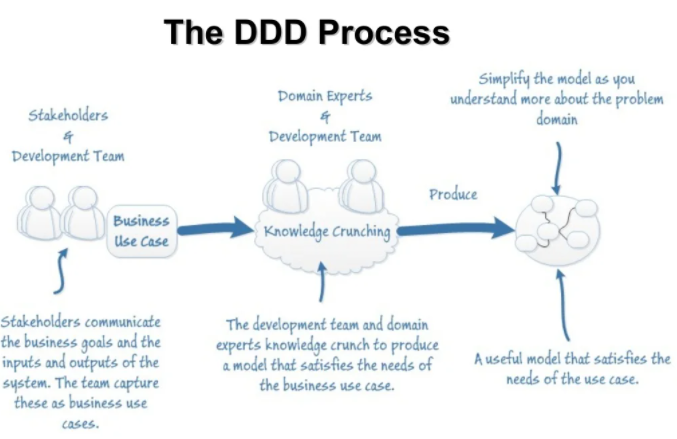
**Ubiquitous Language:** a language structed around the domain model and used by all team members to connect all the activities of the team with the software.

* Use it consistently in speech, documentation, diagrams and code.
* A change in Ubiquitous Language ⬄ change in Model and Code.

**The Model we want:**

* Helps us solve specific problems in our domain.
* Is not necessary “relistic”.
* Forms the basis of a language.
* Should remain current.



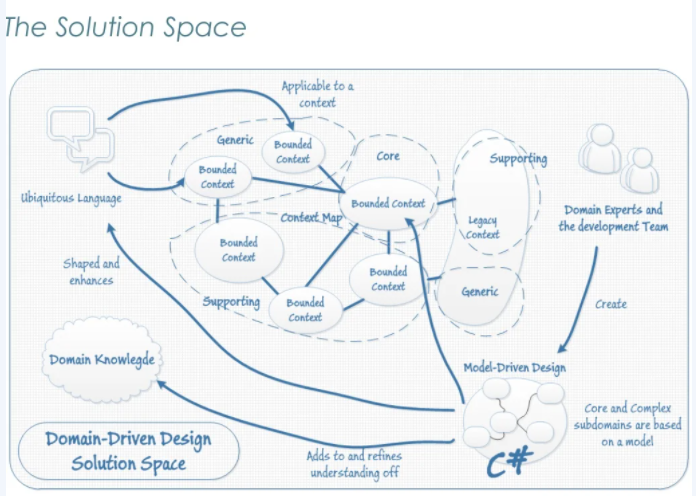


**Domain and Sub Domains:**

In DDD, everything should be communicated under ubiquitous language so the technical team and business team can use the same terms and have same views on the problems

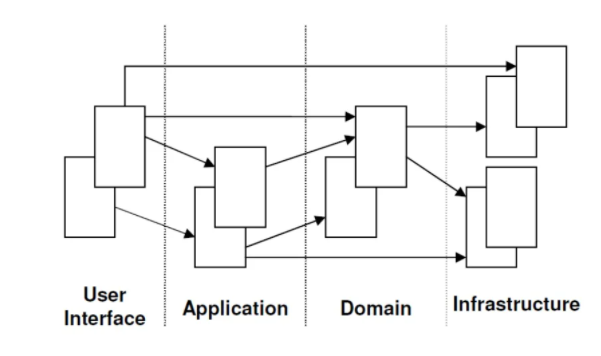
* **Domain** in DDD represent real problem in business. Such as: E commerce is a domain, Payroll system is a domain
* Domain is divided into many **sub domains**, so each sub domains focus smaller problems. Such as: E commerce has many sub domains such as: Shopping Cart, Billing, Product Catalog, Customer Information...
* Each sub domain should have explicit responsibilities so it has a boundary to limit their functionalities, the boundary will help sub domain focus to do only 1 thing and do well. This boundary is considered as **bounded context** of the sub domain. The bounded context will define:
  + How many domain models needed for the sub domain?
  + Which properties needed in the each model?
  + Which functionalities needed in sub domain?

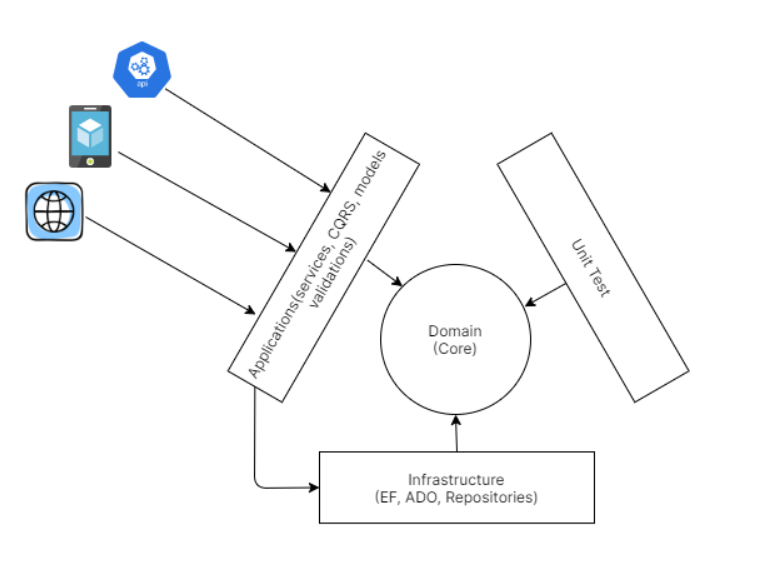
Ex: Shopping Cart sub domain needs models: Cart, Product, Customer Info... and contains functions to perform CRUD on the cart. Notes: The Product and Customer model in the Shopping Cart sub domain maybe not the same with the models in Product Catalogs and Customer Profiles sub domain, they just contain necessary properties to display on Shopping Cart.



**Layers:**

* **User Interface:** Responsible for presenting information to the user and interpreting user commands.
* **Application:** this is a thin layer which coordinates the application activity. It does not contain business logic. It does not hold the state of the business objects.
* **Domain:** this layer contains information about the domain. This is the heart of the business software. The state of the business objects is held here. Persistence details delegated to the infrastructure layer.
* **Infrastructure:** this layer acts as a supporting library for all the other layers. It provides communication between layers, implements persistence for business objects, contains supporting libraries for the user interface layer, etc.





**Model Expressed in Software:**

1. **Entities:**

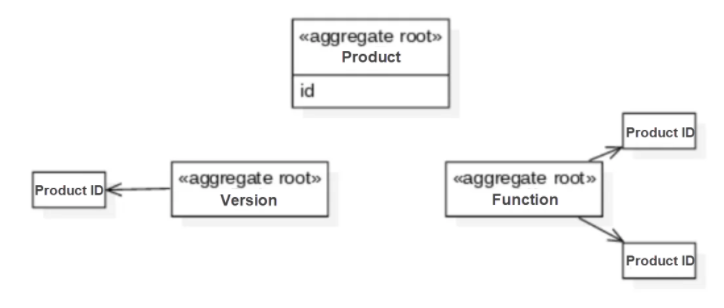
* Objects which have an identity which remains the same throughout the states of the software.
* Must be distinguished from other similar objects having the same attributes (e.g Bank Account for Jim Smith)
* Attributes of an entity can change (**Mutable**)
* Entity should have behavior (**Business Login**)
* No persistence behavior

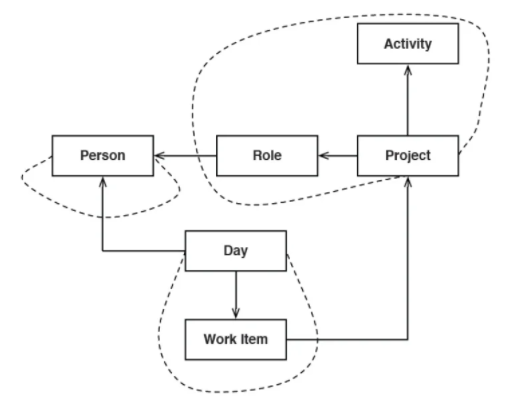
1. **Value Objects:**

* Value Objects are the “things” within your model that have no uniqueness.
* They are only equal to another value object if all their attributes match.
* Value objects are interchangeable.
* Attributes of a value object cannot change, they must be replaced (**Immutable**)
* Example: Money, Address (usually), DTO
* **Entity vs Value Object:**
* Entities have their own intrinsic identity, value objects don’t.
* The notion of identity equality refers to entities; the notion of structural equality refers to value objects; the notion of reference equality refers to both.
* Entities have a history; value objects have a zero lifespan.
* A value object should always belong to one or several entities, it can’t live by its own.
* Value objects should be immutable; entities are almost always mutable.
* To recognize a value object in your domain model, mentally replace it with an integer.
* Value objects shouldn’t have their own tables in the database.
* Always prefer value objects over entities in your domain model.
* **Identifier, structural, Reference equalities:**
* Identifier equality: implies a class has an id field. Two instances of such a class would be equal if they have the same identifiers (id == id).
* structural equality: we consider two objects equal if all of their members match (fields values == fields values)
* Reference equality: means that two objects are deemed to be equal if they reference the same address in the memory (GetHashCode() == GetHashCode())

1. **Aggregates:**

* An Aggregate is a cluster of Entities and Value objects. (Object Graph) Each aggregate is treated as one single unit.
* Each Aggregate has one root entity known as the **Aggregate Root.**
* The root identity is global, the identities of entities inside are local.
* External objects may have references only to root.
* Internal objects cannot be changed outside the Aggregate.
* Aggregation, as a level in domain modeling, enables information hiding, increases the abstraction level, and encapsulates closely related business logic through appropriate boundaries. As a result, the consistency of system data is ensured, and the system performance is improved.
* **Four heuristic rules for Aggregation Recognition:**
  + 1. **Consistency of the lifecycle**:  if the aggregate root disappears, all other elements in the aggregation should disappear at the same time.
    2. **Consistency of the problem domain:**  a new principle that overrides principle 1 will be obtained that says objects that do not belong to the same problem domain should not appear in the same aggregation. (for example article and comment relation if article deleted then its comments will be deleted but if there is books that has comments also so The clear fact is that comment is essentially far from the article )
    3. **Consistency of the scenario frequency:**  Objects that are often operated on at the same time often belong to the same aggregation. Generally, objects that receive very little attention at the same time should not be classified as the same aggregation.
    4. **As few elements as possible within the Aggregation**
* **Aggregation Definitions for Repositories and Factories:**
* Factory mode exists to separate the construction and use of objects.
* Factory design based on aggregation is more important for simplifying the complexity of the system
* Only one factory, which must be visible to the outside, is the aggregation factory.
* The repository is the storage mechanism of aggregation
* He external world can only complete the access to the aggregation through the repository
* An aggregate can have only one repository object in terms of design constraints, which is a repository named after the aggregate root.
* No other objects should be provided with repository objects.





1. **Services:**

* **Domain Service:**  Encapsulates *business logic* that doesn't naturally fit within a domain object, and are **NOT** typical CRUD operations – those would belong to a *Repository*.
* Are Stateless.
* Has to be offered as an interface that is defined as a part of the model, its parameters and results should be domain objects.
* **Application Services:**  Used by external consumers to talk to your system (think Web Services). If consumers need access to CRUD operations, they would be exposed here.
* **Infrastructure Services:**  Used to abstract technical concerns (e.g. MSMQ, email provider, etc).
* Keeping Domain Services along with your Domain Objects is sensible – they are all focused on domain logic. And yes, you can inject Repositories into your Services.

1. **Factories:**

* An object whose responsibility is the creation of other objects.
* Create and manage complex domain objects.
* Especially useful for creating aggregates.

1. **Repositories:**

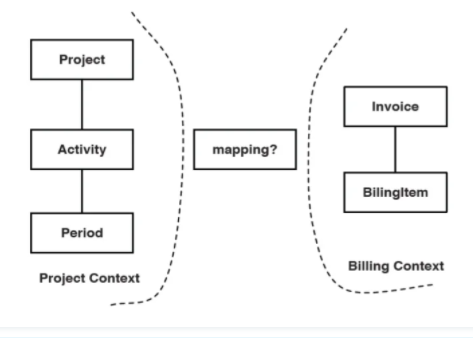
* A Repository encapsulates domain objects persistence and retrieval.
* Clean separation and one-way dependency between the domain and data mapping layers.
* May encapsulate different fetching strategies, distributed caching, NoSQL, Web Service, etc.
* Acts as collection except with more elaborate querying capability.
* One Repository per Aggregate.

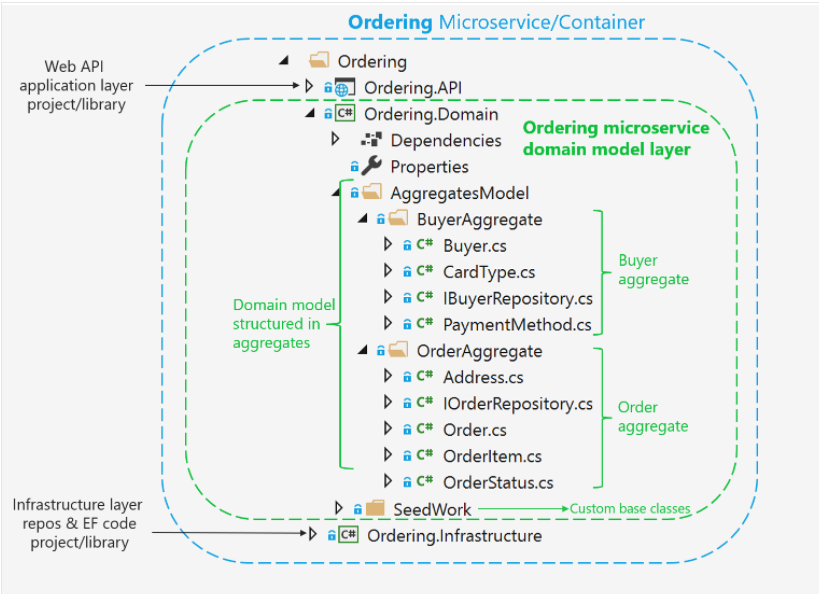
1. **Modules:**

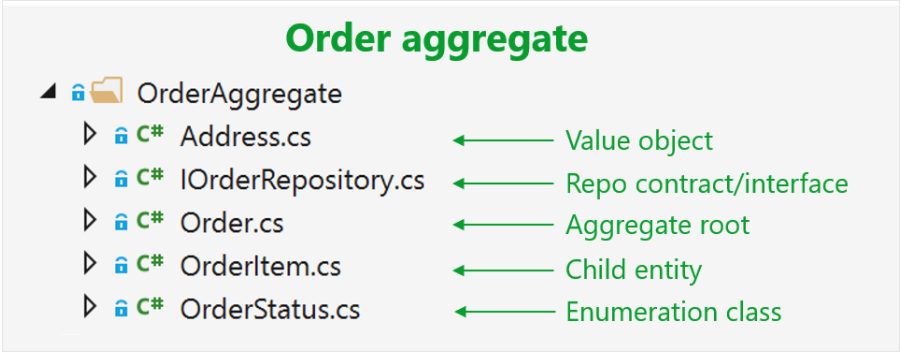
* Break up your domain to reduce complexity
* High cohesion within module, loose coupling between modules.
* Becomes part of the ubiquitous language
* Helps with decoupling
* Aids in extensibility

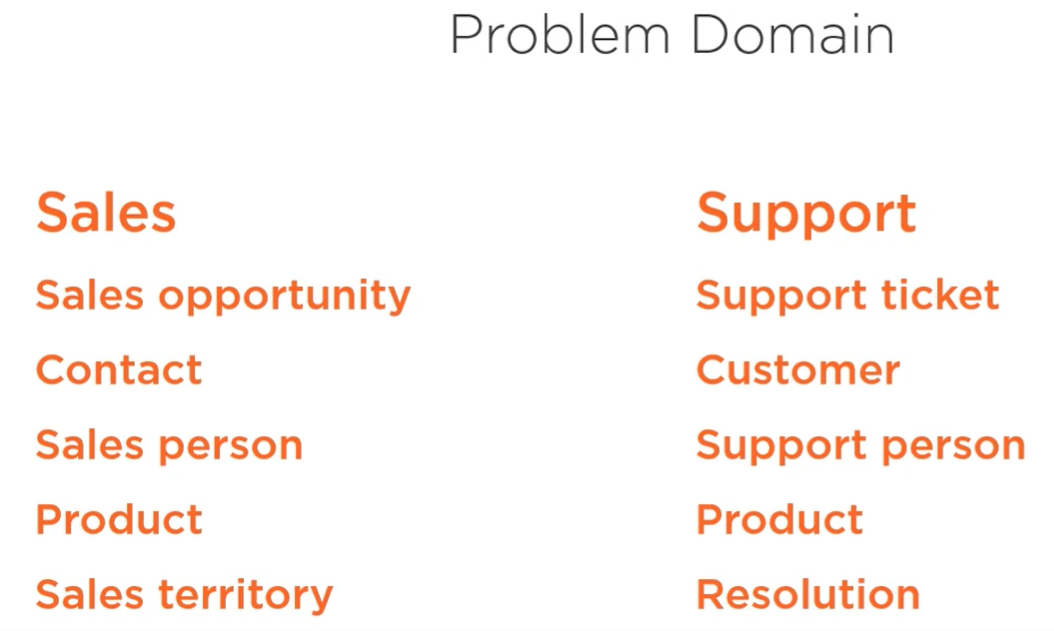
1. **Context Mapping:** Mapping the contract points and translations between bounded contexts

* **Patterns for context mapping:**
  + 1. Shared kernel
    2. Customer/supplier
    3. Confimist
    4. Anti-corruption layer
    5. Separate ways

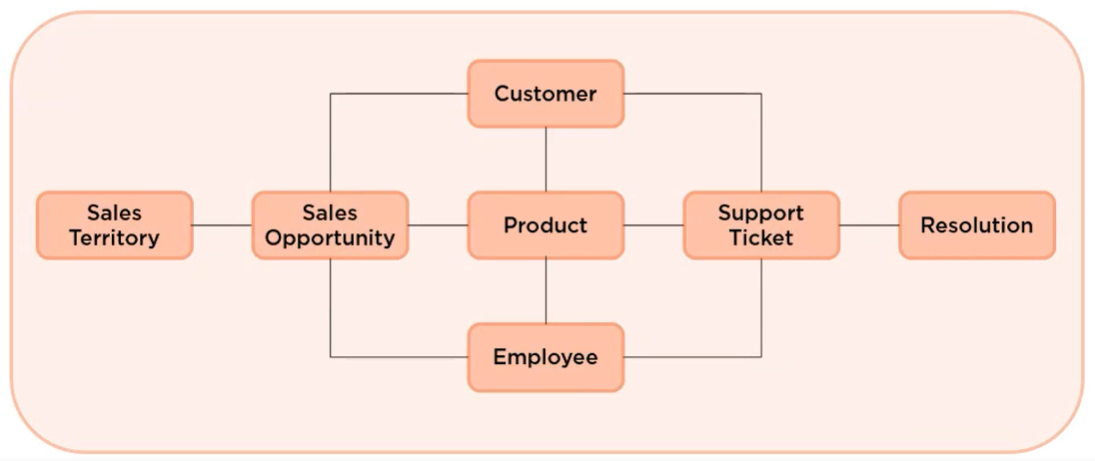




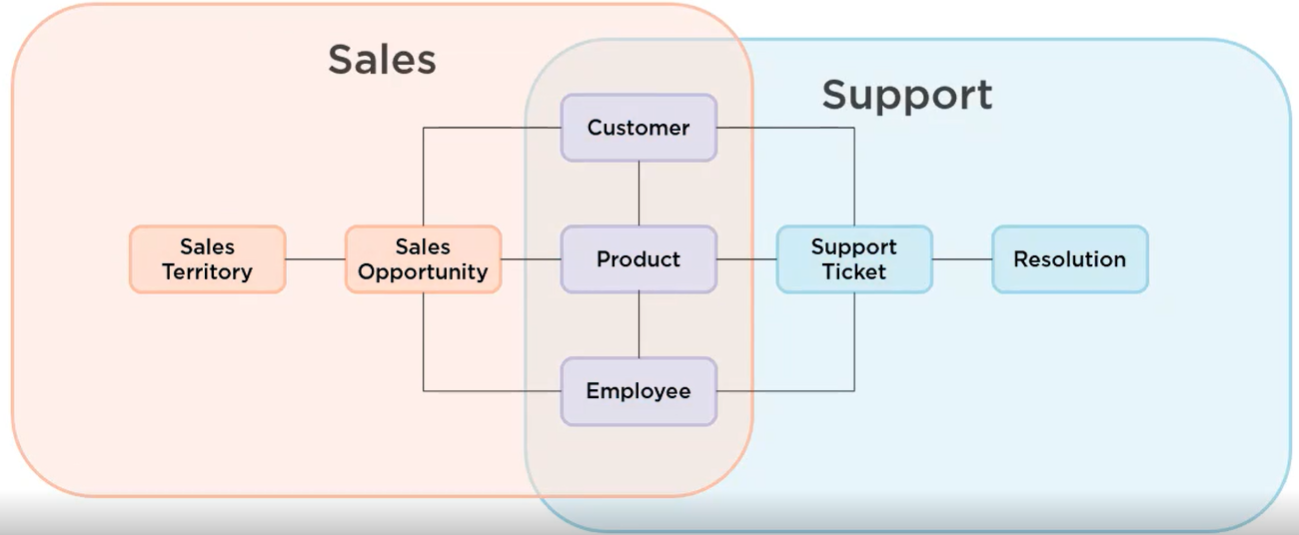




* **Single Domain Model:**



* **Overlapping Context:**



* **Bounded Context:**

