**DDD will help in Enterprise applications because:**

* Amount of data: Low
* Performance: Low
* Business logic complexity: High
* Technical complexity: Low

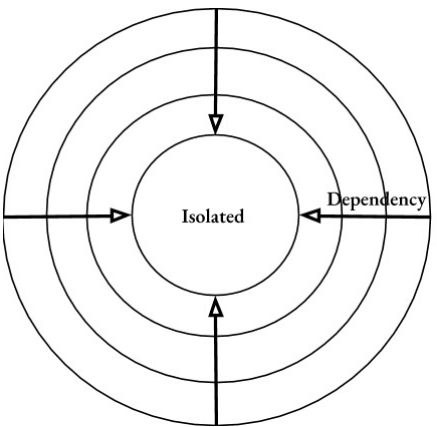
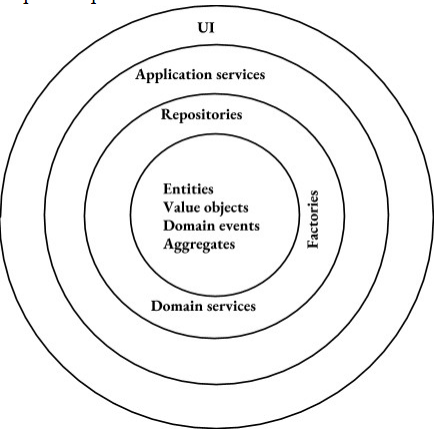
**Core principles in software development:**

* YAGNI (you are not gonna need it)
* KISS (keep it short and simple)

These principles are important, because they help solve two major problems we face when building software projects: shortening the time needed for development, and keeping the code base maintainable in the long run. The most difficult task in the modern business line software is to keep that complexity under control.

1. Ubiquitous Language: Bridges the gap between developers and experts. The concept of ubiquitous language also means you should keep your code base in sync with this single terminology and name all your classes and tables in the database after the terms in the ubiquitous language
2. Bounded Context: Clear boundaries between different parts of the system. Code elements that make sense in one part of the system may seem completely irrelevant in another. In this case, the best solution would be to separate these parts from each other explicitly
3. Core domain: Focus on the most important part of the system. Domain-driven design proposes that we always focus most of our efforts on the core domain. These concepts, ubiquitous language, bounded context, and core domain, are the most important parts of domain-driven design

**Onion Architecture and Domain Isolation**



Upper layers depend on the lower ones, but the lower layers don't know of the upper

These four elements, entities, value objects, domain events, and aggregates, are the most basic. They can refer to each other, for example, and then they can contain a value object or a value object can keep a reference to an aggregate root, but they cannot work with other DDD notions, such as repositories and factories. Similarly, repositories, factories, and domain services can know of each other and the four basic elements, but they should not refer to the application services.

CORE DOMAIN :

* Domain knowledge: YES
* Persistence logic: NO
* Construction logic: NO
* Mapping to the database logic: NO

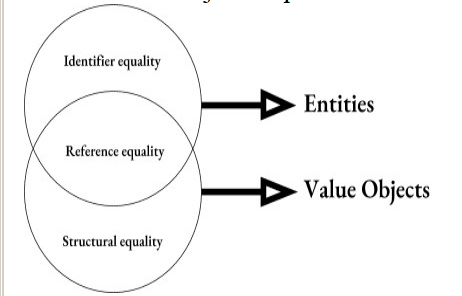
It is crucial to leave entities and value objects to do only one thing, represent the domain logic in your application. In practice, it means they shouldn't contain any knowledge about how they are persisted or how they are created. These two operations must be up to repositories and factories. They also shouldn't contain any knowledge about the tables and columns in the database where they are stored. This must be given away to data mappers. All they should know of is the domain they represent.

**Modeling Best Practices**

* Focus on the core domain first, and pay most of your attention to it. In practice, it means that you should always start the development with modeling the core domain, even if you don't have any UI or database structure yet
* In most enterprise-level applications, the value distribution corresponds to the number of unit tests in this way. The closer we get 100%, the less value the additional tests provide us with. It means that at some point, the value we get from the additional tests doesn't justify the resources we invest in them. In practice, it means you should cover with unit tests only those parts of your code base that are the most significant to the application, and this is the innermost layer in your onion architecture, entities, value objects, aggregates, and domain events, the elements which contain most of the domain. It's a good idea to get 100% or close to 100% as coverage of them. That is another reason why we should keep the core layer of the domain model isolated from other parts of the application

**Module 2: Starting with the First Bounded Context**

* Reference equality, identifier equality, and structural equality.
  + Reference equality means that two objects are deemed to be equal if they reference the same address in the memory.
  + Identifier equality implies a class has an ID field. Two instances of such a class would be equal if they have the same identifiers.
  + Structural equality, we can see there are two objects equal if all of their members match.



* **Entities:**
  + Have inherent Identity
  + Mutable
* **Value Objects:**
  + Don’t have an Id field
  + Can be treated interchangeably
  + Immutable
* The next difference is that value objects cannot live by their own. They should always belong to one or several entities.
* An important implication from this point is that value objects don't have their own tables in the database
* A concept can be an entity in one domain model and a value object in another.

**Prefer value objects to entities:**

* Value objects are light-weight
* Put most of business logic to value objects
* Entities act as wrappers

**Base class**

Use of an interface doesn't show the appropriate relationship between domain entities. Implementing an interface means that your class makes a promise to have some functionality defined in the interface.

“Can-do” relationship:

public interface IEntity{ }

public class Entity1 implements IEntity{ }

public class Entity2 implements IEntity{ }

“Is-a” relationship:

public abstract class Entity{ }

public class Entity1 extends Entity{ }

Value Object bas class

1. Why need override
   1. First, the new two methods are abstract, meaning that we won't forget to implement them in a derived value object class. The compiler will notify us about that.
   2. Second, we are making sure that the object common to the equalsCore method is of the same type as the current valueObject and it is not null. Thus, we don't need to duplicate these checks in the derived classes, we can just gather them here.

* Entity base class
  + Reference equality
  + Identifier equality
  + Should have an identity
  + Single place for equality members
* Value Object base class
  + Reference equality
  + Structural equality
  + Don’t have an identity
  + No single place for equality members

**Unit Test**

we should cover with unit tests only the innermost layer of the onion architecture, entities, value objects, aggregates, and domain events.

* Test First
  + we are pretty sure what we want the code to do, so we can create unit tests up front before we actually start implementing the required functionality
* Code First
  + we might be exploring new areas in our domain model. When experimenting with different ideas in code, we are not exactly sure how the implementation should look like. while experimenting, we often rewrite our code and even throw it away completely, and unit tests would only slow us down with that. If so, that we wrote the first draft of our domain model without any tests.

**Code-first approach for experiments**

**Test-first approach after the experiments**

**Always cover the model with unit tests**

**Module 3: Introducing UI and Persistence Layers**

Controller acts as a mediator between the domain model and the UI. You can think of a Controller as a wrapper. It works on top of one or several entities and allows a View to easily interact with those entities. The two main elements that enable smooth communication between Views and Controller is REST.

UUIDs:

public abstract class Entity{

private UUID id;

private Entity() {

              id= UUID.randomUUID();

}

public UUID getId(){

return id;

}

}

[TableHiloGenerator](https://www.google.com/url?q=https://docs.jboss.org/hibernate/orm/3.5/api/org/hibernate/id/TableHiLoGenerator.html&sa=D&ust=1541229000140000) : need to check again in detail how this works.

**Module 4: Extending the Bounded Context with Aggregates**

Aggregate is a design pattern that helps us simplify the domain model by gathering multiple entities under a single abstraction. This concept includes several implications.

* First of all, an aggregate is a conceptual whole, meaning that it represents a cohesive notion of the domain model. Every aggregate has a set of invariants, which it maintains during its lifetime. It means that in any given time, an aggregate should reside in a valid state.
* kind of validation should be performed in the aggregate so that it's not possible for the client code to add more snacks if the overall weight exceeds the limit
* Every aggregate should have a root. That is, the entity which is the domain for the aggregate, so to speak. An important rule regarding this notion is that classes outside of the aggregate can only reference the root of that aggregate. However, try to avoid exposing the internal entities at all if possible.
* Aggregates also act as a single operational unit for the code in your application layer. Application Services should retrieve them from the database, perform actions, and store them back as a single object. In other words, they should consider an aggregate a conceptual whole and refrain from working with separate entities in it.
* Another function aggregates hold is maintaining consistency boundaries. It means that in any given time the data in the database that belong to a single aggregate should be consistent. To achieve this, we need to persist an aggregate in a transactional manner.
* **Entity can belong to a single aggregate only**
* **Value object can belong to multiple aggregates**
* Entities inside comprise a cohesive group of classes
* Entities in different aggregates should maintain loose coupling among each other

1. Most aggregates consist of 1 or 2 entities
2. 3 entities per aggregate is usually a max
3. The number of Value Objects per aggregate is unlimited

If you find a class in your domain model holding a collection of entities and that collection contains more than, say, 30 members, it's a strong sign you should revisit the model and probably remove the collection and extract the entity on the many side to its own aggregate

The aggregate root base class usually has three goals.

* The first one is to explicitly show the boundaries of the aggregates in your domain model. By inheriting an entity from the AggregateRoot base class, you make it easier to read your code base and see which entities are roots of their own aggregates and which are just part of existing ones.
* Secondly, if you employ optimistic locking, you need to somehow version the entire aggregate. The best way to implement such versioning is to put a version property to the AggregateRoot, like this.
* And finally, the AggregateRoot base class is a perfect place to hold domain events that happen to an aggregate during its lifetime.

It is a good idea to keep the entities that are not aggregates roots inside the boundaries of their aggregates, and not show them to other aggregates

This would be a value object, remove the setters, because our value object will be immutable.

**Aggregates gather multiple entities under a single abstraction**

* Conceptual whole
* Root entity
* Single operational unit for the application layer
* Consistency boundaries
* How to find proper boundaries for aggregates
* Does an entity makes sense by its own?
* Try not to expose internal entities outside the aggregate
* Revealing a hidden abstraction

**Module 5: Introducing Repositories**

Repository is a pattern for encapsulating all communications with the database

The general rule is that there should be a repository per each aggregate.

* one repository per each aggregate
* Repository public methods should accept and return aggregate roots only
* The public methods should work with aggregate roots only. The SnackMachineRepository should accept and return the SnackMachine instances. All the work with internal entities has to be done behind the scenes, either manually in the repository, or using ORM mapping capabilities.
* Repository public methods should accept and return aggregate roots only
  + For example, in our case, if the client code would like to retrieve a slot by its identifier, we could create a method like this one.
  + Repository repository = new SnackMachineRepository();
  + SnackMachine snackMachine = repository.getBySlotId(slotId);
  + You can see this method resides in the SnackMachineRepository, and although it accepts a slotId, it returns an instance of SnackMachine, not Slot.
* it's a good idea to commission all communications with the database to repositories, even if such communications bypass the ORM.
* Define reference data in your domain model explicitly

**Module 6: Introducing the Second Bounded Context**

Bounded context is a central pattern in domain-driven design. It stands for separating the model and explicitly drawing the boundaries between its pieces.

1. First of all, they act as a boundary for the ubiquitous language. It means that the language we use for communicating with domain experts and naming classes in our domain model should be consistent and unified only within a bounded context.
   * At the same time, the naming doesn't have to be consistent across different models. You can think of bounded contexts as if they were Java package for the classes in your code base.
2. they span across all layers in the onion architecture. Each of them is represented with its own onion (onion architecture). So, if you decide to introduce a new bounded context, it should have its own set of entities, repositories, factories, Application Services, and all other layers from the onion architecture
3. A context map is a map that renders the bounded contexts in your system and the connections between them.

sub-domain belongs to the problem space, whereas bounded context to the solution space. In other words, a sub-domain is a part of the whole problem, a part of the problem domain, and bounded contexts at the same time is a part of the solution for that problem. Sub-domains and bounded contexts are best related to each other as 1-to-1, meaning that ideally every sub-domain should be covered by exactly one bounded context. It's not always possible, though.

* the best way to do that is to adhere to the 1-to-1 guideline
* the notion of sub-domain refers to the problem space, it is often defined by customers and domain experts. carefully listen to the domain experts
* If your sub-domain is too big and that causes the team working on it to grow more than, say, 6-8 developers, it's a strong sign you need to separate the bounded context in 2 and form an independent team for each of them
* The general rule of thumb here is that the code of single bounded context should fit your head, meaning that you shouldn't have a lot of trouble understanding it
* shouldn't be a situation where two teams work on a single bounded context

**Types of Physical Isolation**

1. The first one is keeping the bounded context in the same jars, but in different packages, so basically just creating separate folders for them
2. The second type of isolation is extracting the bounded contexts into separate jars under the same solution. Here, for example, you can see two solution folders for bounded contexts, and a separate project for the SharedKernel
3. The third type of isolation is separate deployment. While the first two types imply the bounded context work in a single physical process, this type of isolation means you deploy and maintain them as separate applications. The source code is stored separately and there are separate database instances for each of the bounded contexts. This type of isolation is often referred to as microservices

**Type 1 ------> Type 2 -------> Type 3**

**Easier to maintain proper isolation, bigger maintenance overhead**

**Start with Type 1 isolation, move further only if necessary**

The communication pattern depends on whether or not there is an anti-corruption layer between the bounded contexts.

* If there is none, entities in these bounded contexts can just call other entities' methods directly. Also, the communication can be performed via domain events
* In case there is an anti-corruption layer, things get more complicated, and anti-corruption layer usually means that developers working on one bounded context don't want to interfere with the concepts from another bounded context

Code Reuse

**Business logic :**

* Shouldn’t be reused in most cases
* Extract to a shared kernel

**Domain base classes**

* Reuse within a single team only

**Utility code**

* Reuse within a single team
* Reuse across teams only if provides a lot of value

Overall, try to avoid reusing code between bounded contexts as much as possible, especially reusing the code from the domain layer. Extract a domain class to a shared kernel only when all bounded contexts involved have the same perspective on what it represents.

The distinction between them is that sub-domains belong to the problem space, whereas bounded contexts is the solution for that problem. We discussed the 1-to-1 guideline. You should try to create a single bounded context for each sub-domain in your system. But keep in mind that it's not always possible. There are three reasons why you might want to depart from this practice. Legacy project, large code base, and large team. We talked about the importance of drawing a context map between bounded contexts. The main guideline here is that it should reflect the actual state of affairs, not the desired one. You learned three types of physical isolational bounded contexts. Keep in mind that while your code base is small, it makes sense not to separate bounded contexts physically and just keep them in a single jar. As the project grows, however, consider extracting different bounded contexts out of it to separate to your projects, or even to separate marker services. At the same time, make sure you maintain proper boundaries regardless of what type of physical isolation is chosen. We discussed communication between bounded contexts. It depends on the type of isolation you've chosen, and whether or not there is an anti-corruption layer between the contexts. We also talked about code reuse. The main rule here is that you should avoid reusing code that represents domain logic. In the next module, we will talk about domain events and two different ways of working with them.