# Tutorial Domain Driven Design

**v. 0.1.9-SNAPSHOT**

**Manual**

**2015-05-11**

# Table of Contents



©2015, • ALL RIGHTS RESERVED.

©2015, • ALL RIGHTS RESERVED.

# 1 Introduction DDD

## Introduction Domain Driven Design

The philosophy of Domain Driven Design (DDD), firstly described by Eric Evans, is about placing our attention at the heart of the application, focusing on the complexity that is intrinsic to the business domain itself.

Domain Driven Design consists of a set of patterns for building enterprise applications based on the domain model. Within this introduction we are going to run through some of concepts and terminology of DDD.

### 1.1 Domain Model

At the heart of DDD lies the concept of the domain model. This model is built by the team responsible for developing the system. The team consists of both domain experts from the business and software developers. The role of the domain model is to capture what is valuable or unique to the business. The domain model serves the following functions:

- It captures the relevant domain knowledge from the domain experts.
- It enables the team to determine the scope and verify the consistency of the knowledge.
- The model is expressed in code by the developers.
- It is constantly maintained to reflect evolutionary changes in the domain.

Domain models are typically composed of elements such as entities, value objects, aggregates, and described using terms from a ubiquitous language.

### 1.2 Ubiquitous language

The ubiquitous language is very closely related to the domain model. One of the functions of the domain model is to create a common understanding between the domain experts and the developers. By having the domain experts and developers use the same terms of the domain model for objects and actions within the domain, the risk of confusion or misunderstanding is reduced.

### 1.3 Entities, value objects and services

DDD uses the following concepts to identify some of the building blocks that will make up the domain model:

- Entities are objects that are defined by their identity and that identity continuous through time.
- Value objects are objects which are not defined by their identity. Instead they are defined by the values of their attributes.
- Services is a collection of stateless methods that doesn't model properly to a entity or value object.

### 1.4 Aggregates and aggregate roots

DDD uses the term aggregate to define a cluster of related entities and value objects that form a consistency boundary within the system. That consistency boundary is usually based on transactional consistency. The aggregate root (also known as root entity) is the gatekeeper object for the aggregate. All access to the objects within the aggregate must occur through the aggregate root; external entities are only permitted to hold a reference to the aggregate root. In summary, aggregates are mechanism that DDD uses to manage the complex set of relationships between the many entities and value objects in a typical domain model.

©2015, • ALL RIGHTS RESERVED.

## 1.5 Bounded context

For a large system, it may not be practical to maintain a single domain model; the size and complexity would make it difficult to keep it consistent. To address this, DDD introduces the concept of bounded context and multiple models. Within a system, you might choose to use multiple smaller models rather than a single large model, each focusing on a aspect or grouping of functionality within the overall system. A bounded context is the context for one particular domain model. Each bounded context has its own ubiquitous language, or at least its own dialect of the domain ubiquitous language.

### 1.5.1 Anti corruption layers

Different bounded contexts have different domains models. When your bounded contexts communicate with each other, you need to ensure that concepts specific to one domain do not leak into another domain model. An anti corruption layer functions as a gatekeeper between bounded contexts and helps you to keep the domain models clean.

### 1.5.2 Context maps

A large complex system can have multiple bounded contexts that interact with each other in various ways. A business entity, such as a customer, might exist in several bounded contexts. However, it may need to expose different facets or properties that are relevant to a particular bounded context. As a customer entity moves from one bounded context to another, you may need to translate it so that is exposes the relevant facets or properties for that particular context. A context map is the documentation that describes the relationships between these bounded contexts.

### 1.5.3 Bounded contexts and multiple architectures

Following the DDD approach, the bounded context will have its own domain model and its own ubiquitous language. Bounded contexts are also typically vertical slices through the system, meaning that the implementation of a bounded context will include everything from the data store, right up to the UI. One important consequence of this split is that you can use different implementation architectures in different bounded contexts. Due to this you can use an appropriate technical architecture for different parts of the system to address its specific characteristics.

### 1.5.4 Bounded contexts and multiple development teams

Clearly separating bounded contexts, and working with separate domain models and ubiquitous languages makes it possible to develop bounded contexts in parallel.

### 1.5.5 Maintaining multiple bounded contexts

It is unlikely that each bounded context will exist in isolation. Bounded contexts will need to exchange data with each other. The DDD approach offers a number of approaches for handling the interactions between multiple bounded contexts such as using anti-corruption layers or using shared kernels.

©2015, • ALL RIGHTS RESERVED.

# 2 Building blocks DDD

## Building blocks Domain Driven Design

The diagram below is a navigational map. It shows the patterns that form the building blocks of Domain Driven Design and how they relate to each other.

By using these standard patterns we bring order in the design and make it easier for team members to understand each other's work. Using standard patterns also adds to the *ubiquitous language* which all team members can use to discuss model and design discussions.

©2015, • ALL RIGHTS RESERVED.

# 3 Aggregates

Aggregates.

It is difficult to guarantee the consistency of changes to objects in a model with complex associations. Invariants need to be maintained that apply to closely related groups of objects, not just discrete objects. Yet cautious locking schemes cause multiple users to interfere pointlessly with each other and make a system unusable.

*Therefore*, cluster the `Entities` and `Value Objects` into `Aggregates` and define boundaries around each. Choose one `Entity` to be the root of each `Aggregate`, and control all access to the object inside the boundary through the root. Allow external objects to hold references to the root only. Transient references to internal members can be passed out for use within a single operation only. Because the root controls access, it cannot be blindsided by changes to the internals. This arrangement makes it practical to enforce all invariants for objects in the `Aggregate` and for the `Aggregate` as a whole in any state change.

©2015, • ALL RIGHTS RESERVED.

# 4 Entities

## Entities

*Many object are not fundamentally defined by their attributes, but rather by a thread of continuity and identity.*

Some objects are not defined primarily by their attributes. They represent a thread of identity that runs through time and often across distinct representations. Sometimes such an object must be matched with another object even though attributes differ. Mistaken identity can lead to data corruption.

*Therefore*, when an object is distinguished by its identity, rather than its attributes, make this primary to its definition in the model. Keep the class definition simple and focused on life cycle continuity and identity. Define a means of distinguishing each object regardless of its form or history. Be alert to requirements that call for matching objects by attributes. Define an operation that is guarenteed to produce a unique identity for each object. The model must define what it *means* to be the same thing.

©2015, • ALL RIGHTS RESERVED.

# 5 Factories

## Factories

*When creation of an object, or an entire* `Aggregate`*, becomes complicated or reveals too much of the internal structure* `Factories` *provide encapsulation.*

Creation of an object can be a major operation in itself, but complex assembly operations do not fit the responsibility of the created object. Combining such responsibilities can produce ungainly designs that are hard to understand. Making the client direct construction muddies the design of the client, breaches encapsulation of the assembled object or `Aggregate`, and overly couples the client to the implementation of the created object.

*Therefore*, shift the responsibility for creating instances of complex objects and `Aggregates` to a separate object, which may itself have no responsibility in the domain model but is still part of the domain design. Provide an interface that encapsulates all complex assembly and that does not require the client to reference the concrete classes of the object being instantiated. Create entire `Aggregates` as a piece, enforcing their invariants.

©2015, • ALL RIGHTS RESERVED.

# 6 Layered Architecture

Layered Architecture

In an object oriented program UI, database, and other support code often gets written directly into the business objects. Additional business logic is embedded in the behavior of UI widgets and database scripts. This happens because it is the easiest way to make things work, in the short run. When the domain related code is diffused through such a large amount of other code, it becomes extremely difficult to see and reason about. Superficial changes to the UI can actually change business logic. To change a business rule may require meticulous tracing of UI code, database code, or other programming elements. Implementing coherent model driven objects impractical and automated testing is awkward. With all the technologies and logic involved in each activity, a program must be very simple or it becomes impossible to understand.

**Therefore**:

- Partition a complex program into ’layer’s.
- Develop a design within each ‘layer’ that is cohesive and that depends only on the layers below.
- Follow standard architectural patterns to provide loose coupling to the layers above by means of a mechanism such as *Observer* or *Mediator*; there is never a direct reference from lower to higher.
- Concentrate all the code related to the domain model in one layer and isolate it from the user interface, application, and infrastructure code. The domain objects, free of the responsibility of displaying themselves, storing themselves, managing application tasks, and so forth, can be focused on expressing the domain model. This allows a model to evolve to be rich and clear enough to capture essential business knowledge and put it to work.

A typical enterprise application architecture consists of the following four conceptual layers:

- *Presentation Layer* - Responsible for presenting information to the user and interpreting user commands.
- *Application Layer* - Layer that coordinates the application activity. It doesn’t contain any business logic. It does not hold the state of business objects, but it can hold the state of an application task’s progress.
- *Domain Layer* - This layer contains information about the business domain. The state of business objects is held here. Persistence of the business objects, and possibly their state is delegated to the infrastructure layer.
- *Infrastructure Layer* - This layer acts as a supporting library for all the other layers. It implements persistence for business objects, contains supporting libraries, etc.

## 6.1 Application layer.

The application layer:

- Is responsible for the navigation between the UI screens in the bounded context as well as the interaction with application layers of other bounded contexts.

©2015, • ALL RIGHTS RESERVED.

- Can perform the basic (non business related) validation on the user input data before transmitting it to the other (lower) layers of the application.
- Doesn't contain any business or domain related logic.
- Doesn't have any state reflecting a business use case but it can manage the state of the user session or the progress of a task.
- Contains application services.

## 6.2 Domain layer.

The domain layer:

- Is responsible for the concepts of business domain, and the business rules. Entities encapsulate the state and behavior of the business domain.
- Manages the state of a business use case if the use case spans multiple user requests, (e.g. loan registration process which consists of multiple steps: user entering loan details, system returning products and rates, user selecting particular product, system locking the loan for selected rate).
- Contains domain services.
- Is the heart of the bounded context and should be well isolated from the other layers. Also, it should not be dependent on the application frameworks used in the other layers, (Hibernate, Spring, etc).

## 6.3 CRUD operations

TODO: see Domain Driven Design (DDD) architecture layer design for CRUD operation

©2015, • ALL RIGHTS RESERVED.

# 7 Modules

## Modules

Everyone uses `Modules` but few treat them as full fledged part of the model. Code gets broken down into all sort of categories, from aspects of the technical architecture to developers work assignments. It is a truism that there should be low coupling between `Modules` and high cohesion within them. Explanations of coupling and cohesion tend to make them sound like technical metrics, to be judged mechanically based on the distributions of associations and interactions. Yet it isn't just code being divided into `Modules`, but concepts. There is a limit to how many things a person can think about at once (hence low coupling).

*Therefore*, choose `Modules` that tell the story of the system and contain a cohesive set of concepts. Seek low coupling in the sense of concepts that can be understood and reasoned about independently of each other. Refine the model until it partitions according to the high level domain concepts and the corresponding code is decoupled as well. Give the `Modules` names that become part of the ubiquitous language. `Modules` and their names should refelect insight into the domain.

©2015, • ALL RIGHTS RESERVED.

# 8 Repositories

Repositories.

A client needs a practical means of acquiring references to preexisting domain objects. If the infrastructure makes it easy to do so, the developer of the client may add more traversable associations, muddling the model. On the other hand, they may use queries to pull the exact data they need from the database, or to pull a few objects rather than navigating from the ‘Aggregate’ roots. Domain logic moves into queries and client code, and the `Entities` and `Value Objects` become mere data containers. The sheer technical complexity of applying most database access infrastructure quickly swamps the client code, which leads developers to dumb down the domain layer, which makes the model irrelevant.

*Therefore*, for each type of object that needs global access, create an object that can provide the illusion of an in memory collection of all objects of that type. Setup access through a well known global interface. Provide methods to add and remove objects, which will encapsulate the actual insertion and removal of data in the store. Provide methods that select objects based on some criteria and return fully instantiated objects of object whose attribute values meet the criteria, thereby encapsulating the actual storage and query technology. Keep the client focused on the model, delegating all object storage and access to the `Repositories`.

©2015, • ALL RIGHTS RESERVED.

# 9 Services

Services.

Some concepts from the domain aren't natural to model as objects. Forcing the required domain functionality to be the responsibility of an `Entity` or `Value Object` either distorts the definition of a model based object or adds meaningless artificial objecs.

*Therefore*, when a significant process or transformation in the domain is not a natural responsibility of an `Entity` or `Value Object`, add an operation to the model as a standalone interface declared as a `Service`. Define the interface in terms of the language of the model and make sure the operation name is part of the ubiquitous language. Make the service stateless.

## 9.1 Service types.

Services exist in most layers of the DDD layered architecture:

- Application
- Domain
- Infrastructure

An infrastructure service would be something that communicates directly with external resources, such as file systems, databases, etc.

Domain services are the coordinators, allowing higher level functionality between many different smaller parts. Since domain services are first-class citizens of the domain model, their names and usage should be part of the ubiquitous language. Meanings and responsibilities should make sense to the stakeholders or domain experts.

The application service will be acting as façade and accepts any request from clients. Once the request comes, based on the operation, it may call a Factory to create domain object, or calls repository service to re-create existing domain objects. Conversion between, DTO (Data Transfer Object) to domain objects and Domain objects to DTO will be happening here. Application service can also call another application service to perform additional operations.

The differences between a domain service and an application service are subtle but critical:

- Domain services are very granular where as application services are a facade with as purpose providing an API.
- Domain services contain domain logic that can't naturally be placed in an entity or value object, whereas application services orchestrate the execution of domain logic and don't themselves implement any domain logic.
- Domain service methods can have other domain elements as operands and return values whereas application services operate upon trivial operands such as identity.
- Application services declare dependencies on infrastructural services required to execute domain logic.
- Command handlers are a flavor of application services which focus on handling a single command typically in a CQRS architecture.

©2015, • ALL RIGHTS RESERVED.

# 10 Value Objects

## Value Objects

*Many objects have no conceptual identity. These objects describe some characteristic of a thing.*

Tracking the identity of `Entities` is essential, but attaching identity to other objects can hurt system performance, add analytical work, and muddle the model by making all object look the same. Software design is a constant battle with complexity. We must make distinctions so that special handling is applied only where necessary. However if we think of this category of objects as just the absence of identity , we haven't dded much to our toolbox or vocabulary. In fact, these objects have characteristics of their own, and their own significance to the model. These are the objects that describe things.

*Therefore*, when you care only about the attributes of an element of the model, classify it as a `Value Object`. Make it express the meaning of the attributes it conveys and give it related functionality. Treat the `Value Object` as immutable. Don't give it any identity and avoid the design complexities necessary to maintain `Entities`.

©2015, • ALL RIGHTS RESERVED.

# 11 Glossary

Glossary

## 11.1 A

- **ACL** - See Anti Corruption Layer.
- **Aggregate** - A cluster of associated objects that are treated as a unit for the purpose of data changes. External references are restricted to one member of the `Aggregate`, designated as the root. A set of consistency rules applies within the `Aggregate`'s boundaries.
- **Aggregate root** - Gatekeeper object for the aggregate. All access to the objects within the aggregate must occur through the aggregate root; external entities are only permitted to hold a reference to the aggregate root.
- **Anti Corruption Layer** - Anti-corruption Layers help protect against corruption in the domain model when the system interacts with other systems or bounded contexts. The Anti-corruption Layer translates to and from the other system's models. It is often implemented using the Facade and/or Adapter design pattern.
- **Analysis pattern** - A group of concepts that represents a common construction in business modeling. It may be relevant to only 1 domain or may span many domains (resource fowler 1997, p. 8)
- **Application Layer** - Manage transactions, coordinates application activities, creates and accesses domain objects.
- **Application Service** - Used by external consumers to talk to your system. If consumers need access to CRUD operations, they would be exposed via a application service. Application Services will typically use both Domain Services and Repositories to deal with external requests.
- **Assertion** - A statement of the correct state of a program at some point, independent of how it does it. Typically a `Assertion` specifies the result of an operation or the required input for and operation.

## 11.2 B

- **BC** - See Bounded Context.
- **Bounded Context** - A division of a larger system that has its own ubiquitous language and domain model. Bounding contexts gives team members a clear and shared understanding of what has to be consistent and what can be developed independently.

## 11.3 C

- **Client** - A program element that is calling the element under design, using its capabilities.
- **Cohesion** - Logical agreement and dependence.
- **Command** - An operation that causes some change to the system (for example, setting a variable).
- **Context** - The setting in which a word or statement appears that determines its meaning.
- **Context Map** - A representation of the Bounded Contexts involved in a project and the actual relationships between them and their models.
- **Core Domain** - The distinctive part of the model, central to the user's goals, that differentiates the application and makes it valuable.

©2015, • ALL RIGHTS RESERVED.

## 11.4 D

- **Data Transfer Object** - A Data Transfer Object (DTO) is a data container which is used to transport data between layers and tiers. It mainly contains of attributes. DTOs are anemic in general and do not contain any business logic.
- **DDD** - See Domain Driven Design.
- **Declarative Design** - A form of programming in which a precise description of properties actually controls the software.
- **Domain Driven Design** - Domain Driven Design is not a technology or a methodology. DDD provides a structure of practices and terminology for making design decissions that focus and accelerate software projects dealing with complicated domains.
- **Domain Layer** - Contains the state and behavior of the domain.
- **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.
- **DTO** - See Data Transfer Object.

## 11.5 E

- **Entities** - Entities are characterized by having an identity that's not tied to their attribute values. All attributes in an entity can change and it's still the same entity. Conversely, two entities might be equivalent in all their attributes, but will still be distinct.
- **Event** - An event represents something that took place in the domain. Since an event represents something in the past, it can be considered as a fact and used to take decisions in other parts of the system.

## 11.6 I

- **Infrastructural Layer** - Supports all other layers, includes implementation of repositories, adapters, frameworks, etc.
- **Infrastructure Services** - Used to abstract technical concerns (e.g. MSMQ, email provider, etc)

## 11.7 M

- **Model** - A useful approximation to the problem at hand.
- **Modifier** - See Command.

## 11.8 R

- ** Root entity** - See aggregate root.

## 11.9 S

- **Saga** - In essence, sagas listen for events and dispatch commands. A saga spans more than one messages and manages a process.

## 11.10 U

- **Ubiquitous Language** - A language structured around the domain model and used by all team members to connect all activities of the team with the software. The idea is to avoid translations because they blunt communication and makes knowledge crunching anemic.

©2015, • ALL RIGHTS RESERVED.

- **User Interface Layer** - Accepts user commands and presents information back to the user.

## 11.11 V

- **Value Object** - Value objects have no seperate identity; they are defined solely by their attribute values. Though we are typically talking of objects when referring to value types, native types are actually a good example of value types. It is common to make value types immutable.

©2015, • ALL RIGHTS RESERVED.