



CHAPTER

4

The Domain Model Layer



I.- THE DOMAIN

This section describes the domain logic layer architecture (business rules) and contains key guidelines to be considered when designing this layer.

This layer should be responsible for representing business concepts, information on the status of the business processes and implementation of domain rules. It should also contain states reflecting the status of business processes, even when the technical storage details are delegated to the lower layers of the infrastructure (Repositories, etc.)

The 'Domain Model' Layer is the Heart of the Software.

The components of this layer implement the system core functionality and encapsulate all the relevant business logic (generally called Domain logic according to DDD terminology). Basically, this layer usually contains classes which implement the domain logic within their methods, although it can also be implemented using a dynamic business rule system, etc.

Following the architecture patterns in DDD, this layer must completely ignore the data persistence details. These persistence tasks should be performed by the infrastructure layer.

The main reason for implementing the domain logic layer (business) is to differentiate and clearly separate the behavior of the domain rules (business rules that are the responsibility of the domain model) and the infrastructure implementation details (like data access and specific repositories linked to a particular technology such as O/RMs, or simply data access libraries or even cross-cutting aspects of the architecture). Thus, by isolating the application Domain, we will drastically increase the maintainability of our system and we could even replace the lower layers (data access, O/RMs, and databases) with low impact to the rest of the application.

In each chapter of this guide, the intention is to show the approach on two separate levels. A first logical level (Logical architecture, as in this chapter) that could be implemented with any technology and language (any .NET version or even other non-Microsoft platforms) and subsequently a second level of technology implementation, where we will show how to develop this layer, particularly with .NET 4.0 technologies.



2.- DOMAIN LAYER: LOGICAL DESIGN AND ARCHITECTURE

This chapter is organized into sections that include the domain logic layer design as well as the implementation of the functionalities that are typical of this layer, such as **decoupling** from the data access infrastructure layer using **IoC** and **DI**. It also shows typical concerns within this layer regarding security, cache, exceptions handling, *logging* and validation concepts.

In this diagram we show how this Domain model layer typically fits into our '*N-Layer Domain Oriented*' architecture.

DDD N-Layered Architecture

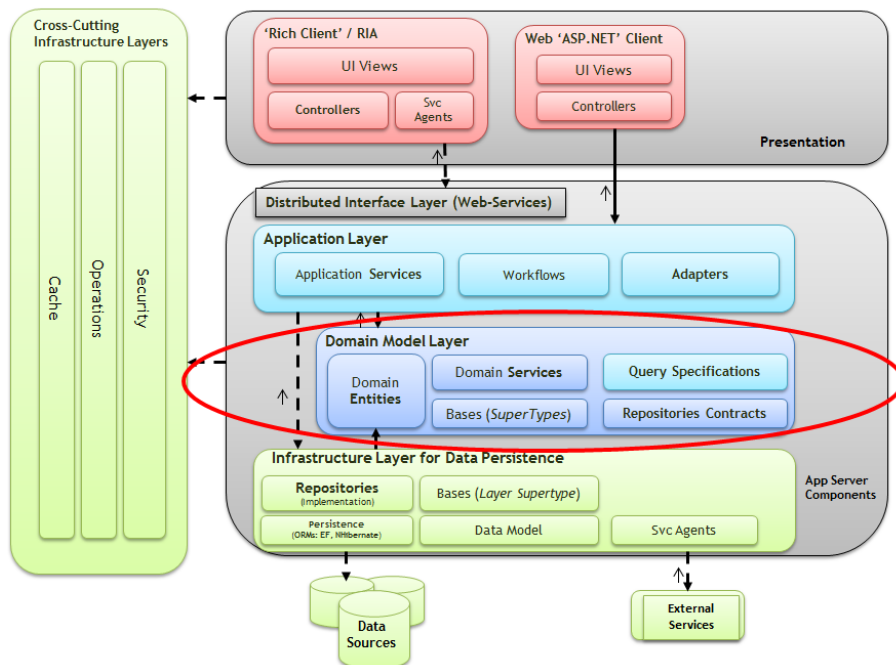


Figure 1.- Position of the Domain Layer within the DDD N-Layered Architecture



2.1.- Sample Application: Business Requirements of a Sample Domain Model to be Designed

Before proceeding with the details of each layer and how to design each one internally, we want to introduce a specific “*StoryScript*” as a ‘*Sample Domain Model*’ which will be designed following the domain oriented design patterns, in layers, and will be implemented at a later time (See our sample application at CODEPLEX.).

Note:

We have defined very simple business requirements below. Their functionality is intentionally very simple, especially in the areas related to banking. This is because the main purpose of the sample application is to highlight aspects of the architecture and design and not to design and implement a real and functionally complete application.

The initial details of domain requirements/problems, at a functional level have been obtained through discussions with domain experts (end users with—expertise in a particular functional area) and are as follows:

- 1.- *A **Customer** and **Order** management application is required. There should also be a **Banking** module related to the Bank of the company in order to make **transfers** and other bank operations for the customers.*
- 2.- *‘Customer lists’ with flexible filters are required. The operators managing the customers need to be able to perform **customer searches in a flexible manner**; being able to search by a part/initial of a name which could be extended in the future to allow searches by other **different attributes** (Country, State/Province, etc.). It would also be very useful to have queries for customers whose orders are in a certain state (e.g. “unpaid.”). The result of these searches is simply a customer list showing their primary data (ID, name, location, etc.).*
- 3.- *An ‘Order list’ owned by a specific customer is required. The total value of each order should be seen on the list, as well as the date of the order and the name of the parties involved.*
- 4.- *An order can have an unlimited number of detail lines (order items). Each order line describes an order item which consists of a product and the required quantity of that product.*
- 5.- *It is important to detect concurrency conflicts.*

IT Department told us that it is acceptable and even advisable to use “Optimistic Concurrency Control”. For example, consider a scenario where a user tries to update data on a data set he retrieved earlier, but another user has modified the original data in the database after the first user retrieved it and just before he saves his changes back to the database. In this case, when the first user tries to update the data, this conflict is detected (original data has been modified and now there’s a possibility of losing those updated data if we save new data over it). Only conflicts causing actual inconsistencies will be considered.

- 6.- An order cannot have a total value less than \$6 nor more than \$1 million.*
- 7.- Each order and each customer should have a user friendly number/code. This code should be legible, writable and easily remembered, as well as having the ability to search for a customer/order using the codes. If necessary, the application could manage more complex IDs but should be transparent to the end user.*
- 8.- An order always belongs to a customer; an order line always belongs to an order. There cannot be orders without a specific customer nor can there be order lines without an order.*
- 9.- Bank operations may be independent from the customers and orders module. They should provide a basic view, such as a list of existing accounts with their relevant data (e.g.. balance, account number, etc.) as well as the ability to perform simplified bank transfers between these accounts (source account and destination account).*
- 10.-The effective implementation of a bank transfer (in this case, persisting timely changes on the account balance stored in the database) should be accomplished in an atomic operation (‘all or nothing’). Therefore, it should be an atomic transaction.*
- 11.-The accounts will have a blocked/unblocked state at the business level. The application manager should be able to unblock/block any chosen account.*
- 12.-If an account is blocked, no operations can be performed against it (no transfers or any other type of operations). If any operation is attempted against a blocked account, the application should detect it and show a business exception to the application user, informing him of the reason why he cannot perform such an operation (for instance, because a specific account is blocked at the business level).*
- 13.- (SIMPLIFICATION OF THE SAMPLE) The intention is to have the simplest sample possible at the functional and data design levels, in order to clearly show and understand the architecture, so the priority should be simplicity in logical entities and database designs. For example, a customer, organization*

and address all merged into the same logical entity and even into the same database table is not the best design at all. However, in this case (sample application) the goal is to end up with a design that maximizes the simplification of the application functionality. In fact, this sample application intends to show the best practices in Architecture, not in logical and database design for application-specific functionality. So, in the unreal world of this application, these characteristics must be considered when simplifying the design:

- *A Customer/Company will only have one contact person (Although this is not true in the real world).*
- *A Customer/Company will only have one address (Although this is usually not the case in the real world because they may have several addresses for different purposes, etc.)*

Based on these specifications, we will identify the specific elements of the sample application such as Entities, Repositories, Services, etc., as we go over the different elements of the Architecture.



2.2.- Domain Layer Elements

During current chapter, we briefly explain the responsibilities of each type of element proposed for the Domain Model:



2.2.1.- Domain Entities

This concept represents the implementation of the *ENTITY* pattern.

ENTITIES represent domain objects and are primarily defined by their identity and continuity over time and not only by the attributes that comprise them.

Entities normally have a direct relationship to the main business/domain objects, such as customers, employees, orders, etc. Therefore, it is quite common to persist such entities in databases, although this depends entirely on each specific application. It is not mandatory but the aspect of "continuity" is usually strongly related to the storage in databases. Continuity means that the entity should be able to "survive" the execution cycles of the application. Each time the application is restarted, it should be possible to reconstruct these entities in memory.

In order to distinguish one entity from another, the concept of identity that uniquely identifies entities is essential, especially when two entities have the same values/data in their attributes. Identity in data is a fundamental aspect in applications. A case of wrong identity in one application can lead to data corruption problems or program

mistakes. Many items in the real domain (the business reality) or in the application domain model (business abstraction) are defined by their identity and not by their attributes. A good example of an entity is a person. The entity's attributes, such as address, financial data or even its name may change throughout its lifetime; however, the entity's identity will remain the same, the same person, in this case. Therefore, the essential concept of an ENTITY is a continuous abstract life that can evolve to different states and shapes, but that will always have the same identity.

Some objects are not defined primarily by their attributes; they represent a thread of identity with a specific life and often with different representations. An entity should be able to be distinguished from other different entities even when they have the same descriptive attributes (e.g., there can be two people with the same first and last names).

With regard to DDD, and according to Eric Evans' definition, "*An object primarily defined by its identity is called ENTITY.*" Entities are very important in the Domain model and they should be carefully identified and designed. What may be an ENTITY in some applications may be not in other applications or even in other BOUNDED-CONTEXT. For example, an "address" in some systems may not have an identity at all, since it may only represent the attributes of a person or company. However, in other systems such as an application for a Power Utility company, the customer's addresses could be very important and therefore the address must have an identity because the billing system can be directly linked to the addresses. In this case, an address should be classified as a Domain ENTITY. In other cases, such as in an e-commerce application, the address may simply be an attribute of the person's profile. In this last case, the address is not so important and should be classified as a VALUE-OBJECT, (as it is called in DDD and we will explain later on.)

An ENTITY can be of many types, it can be a person, car, bank transaction, etc., but the important point is that whether it is an entity or not depends on the specific domain model that we are dealing with. A particular object does not need to be an ENTITY in all application domain models. Also, not all the objects in the domain model are an Entity.

For example, in a bank transaction scenario, two identical incoming amounts on the same day are considered to be different bank transactions, so they have an identity and usually are ENTITIES. Even if the attributes of both entities (in this case, amount and time) were exactly the same, they would still be known as different ENTITIES.

Entity Implementation Design

Regarding **design and implementation**, these entities are disconnected objects (with their own data and logic) and are used to obtain and transfer entity data between different layers. These objects represent real world business entities, such as products or orders. On the other hand, the entities the application uses internally are data structures in memory, such as the classes (entity data **plus** entity logic). Furthermore, if these entities depend on a particular data persistence technology (e.g., prescriptive *Entity Framework entities*), then these classes should be located inside the data persistence infrastructure layer because they are related to a specific technology. **On**

the other hand, if we follow the patterns recommended by DDD and use POCOs (*Plain Old CLR Objects*), they are plain classes (our own code) which are not tight to any particular technology. Therefore, these ENTITIES should be located as elements within the Domain Layer, since they are Domain entities and independent from any infrastructure technology (ORMs, etc.).

Table 1.- Principle of Persistence Technology Ignorance

PI Principle (Persistence Ignorance), POCO and STE

This concept, which recommends **POCO** (*Plain Old CLR Objects*) for domain entity implementation, is probably the most important point to consider when implementing entities according to a Domain Oriented Architecture. It is fully supported by the principle that all components of the Domain layer must completely ignore technologies that the Data Persistence Infrastructure Layer is based on, such as O/RMs.

Persistence Ignorance is a design principle which suggests that the underlying object hide the details of how it is persisted so you can ignore it and work with a simple object. The term “ignorance” suggests that this behavior be encapsulated which is a tenet of Object Oriented design.

The way these entity objects are implemented is especially important for many designs. In those designs (such as in DDD), it is vital to isolate these elements from any knowledge on the data access base technologies, so that they really know nothing about the underlying technology to be used for their persistence (*Persistence Ignorance* principle). In other words, entities that do not inherit from any base class and do not implement any interface related to the underlying technologies are called **POCO** in .NET, or **POJO** (*Plain Old Java Object*) in the Java world.

On the contrary, objects that do inherit/implement a certain base class or interface related to the underlying technologies are known as “*Prescriptive Classes*”. The decision to choose one or the other is not trivial and must be carefully considered. On one hand, using POCO give us a great degree of freedom with respect to the persistence model we choose. On the other hand, however,

it brings restrictions and/or overloads associated with the “degree of ignorance” the persistence engine will have regarding these entities and their correlation to the relational model (this usually involves a higher degree of development efforts). POCO classes have a higher initial implementation cost, unless the O/RM we are using helps us mapping the concepts and even generating code.

The concept of **STE** (*Self Tracking Entities*) is a bit more lax. That is to say, the data classes defined by the entities are not entirely “plain” but rather depend on **implementing one or more interfaces** that specify the minimum implementation to be provided. In this case, it does not completely meet the PI (Persistence Ignorance) principle and it is important for this interface to be under our control (our own code). In other words, the interface must not be part of any external infrastructure technology. Otherwise, our entities would stop being “agnostic” regarding the Infrastructure layers and external technologies and would become “Prescriptive Classes”.

In any case, ENTITIES are objects that float throughout the whole architecture or at least on the Application Server layers. The latter case is when we use DTOs (*Data Transfer Objects*) for remote communications between *Tiers*, where the domain model internal entities would not flow to the presentation layer or any other point beyond the internal layers of the Service. DTO objects would be provided to the presentation layer in a remote location. The analysis of DTOs versus disconnected Entities is covered in the Distributed Services chapter, since these concepts are related to the N-Tier applications and distributed development.

Finally, we must consider the classes' serialization requirements that can exist when dealing with remote communications. Passing entities from one layer to another (e.g., from the Distributed Services layer to the Presentation Layer) will require such entities to be serialized; they will have to support certain serialization mechanisms, such as XML format or binary format. To this effect, it is important to confirm that the chosen entity type effectively supports serialization. Another option is, as we said, conversion to and/or aggregation of DTOs in the Distributed Services layer and Application layer.

Entity Logic contained within the Entity itself

It is essential that the ENTITY objects themselves possess certain logic related to the entity data (data in memory). For example, we can have business logic in a "*BankAccount*" entity, which is executed when money is added or when a payment is made, or even when the amount to be paid is checked (logically, this amount must be greater than zero). The logic of a calculated-field could be another example, and ultimately, any logic related to the internal part of such entity.

Of course, we could also have entities that do not have their own logic, but this case will occur only if we really don't have any related internal entity logic, because if all of our entities had a complete lack of logic, we would be falling into the '*Anemic Domain Model*' anti-pattern introduced by Martin Fowler. See '*AnemicDomainModel*' by Martin Fowler:

<http://www.martinfowler.com/bliki/AnemicDomainModel.html>

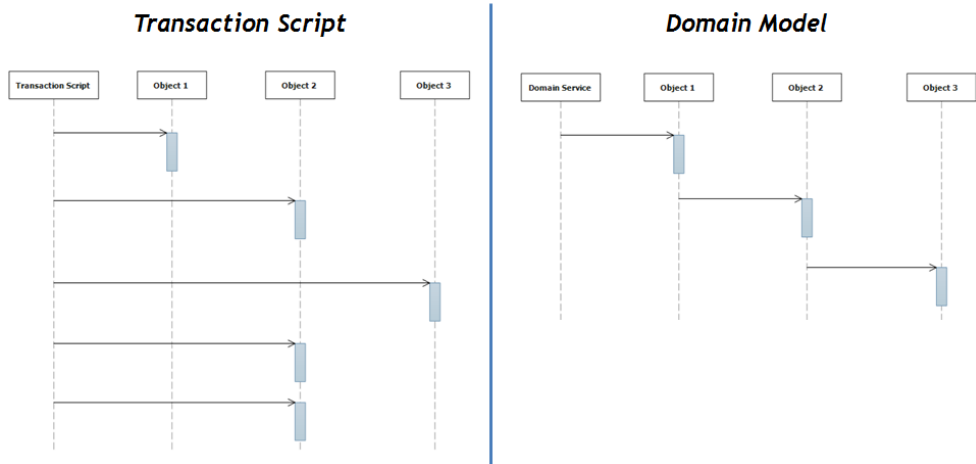
The '*Anemic-Domain-Model*' anti-pattern occurs when there are only data entities such as classes that merely have fields and properties and the domain logic belonging to such entities **is mixed with higher level classes (Domain Services or Application Services)**. It is important to note that, under normal circumstances, Domain Services should not contain any internal entities logic, but a coordination logic that considers such entities as a whole unit or even sets of such units.

If the SERVICES (Domain Services) possessed 100% of the ENTITIES logic, this mixture of domain logic belonging to different entities could be dangerous. That would be a sign of '*Transaction Script*' implementation, as opposed to the '*Domain Model*' or domain orientation.

The '*Transaction Script*' approach has an '*Anemic Model*', because if all the logic is within the services, what are our domain entity objects?. It will be nothing but

structures carrying data. The client code asks the domain to provide data, and then the client code makes decisions based on that data.

The transaction script and only Business-Services layer approach is simple to begin with, but as the application becomes more complex it leads to a bunch of problems, especially as developers try to avoid duplication.



What we achieve by applying the DDD patterns is the elimination of those transaction scripts and many business services. The transaction script (e.g. ASP.NET methods in a web application) will be simply responsible for finding an appropriate entry-point into the domain, then telling that domain class to do some work.

The consequence is that business logic can be expressed more deeply in your domain. You are free to find the right abstractions that will allow you to make your code understandable, and to create small classes with single responsibilities.

This also means that the creation and persistence of objects will be abstracted and encapsulated, without the calling client code or script having to be aware of data access. This is because Domain entities should be 'Persistence Ignorant', shouldn't be allowed to hold references to any kind of DAOs neither to the infrastructure technology used (O/RM libraries).

None of this work has to be done in a transaction script or script – move that responsibility into a place in your domain where it makes sense.

In addition, the **logic related to using/invoking Repositories**, is the logic that should normally be situated within the Application Layer SERVICES. Unless we really need it, it should not be within the Domain Services. An object (ENTITY) does not need to know how to save/build itself, like an engine in real life that provides engine capacity, but does not manufacture itself, or a book does not “know” how to store itself on a bookshelf. This is why we shouldn't invoke Repositories from within the Entities themselves.

The logic we should include within the Domain Layer should be the logic that we could be speaking about with a Domain/Business Expert. In this case, we usually do not talk about anything related to Repositories or transactions with a Domain/Business Expert.

Regarding **using Repositories from Domain Services**, there are, of course, **exceptions**. That is the case when we need to obtain data depending on Domain logic states. In that case we will need to invoke Repositories from the Domain Services. But **that use will usually be just for querying data**. All the transactions, UoW management, etc. should usually be placed within Applications Services.

Table 2.- Framework Architecture Guide rules



Identifying ENTITIES Based on Identity

Rule

- When an object is distinguished by its identity and not by its attributes, this object should be fundamental in defining the Domain model. It should be an ENTITY. It should maintain a simple class definition and focus on continuity of the life cycle and identity. It should be distinguishable in some way, even when it changes attributes or even in form or history. Related to this ENTITY (but probably, defined out of the entity class), there should be an operation ensuring that we can get a single result for each object, possibly selecting a unique identifier. The model must define what it means to be the same ENTITY object.



References

- *'ENTITY pattern'* in the book *'Domain Driven Design'* by Eric Evans.
- **The Entity Design Pattern**
- <http://www.codeproject.com/KB/architecture/entitydesignpattern.aspx>

Table 3.- Framework Architecture Guide Rules



ENTITIES in a Domain Oriented Architecture should be either POCO or STE.

○ **Rule**

- In order to meet the **PI (Persistence Ignorance)** principle and not to have direct dependencies to infrastructure technologies, it is important that our entities being implemented as POCO or STE (Productive way for N-Tier apps but a more lax way regarding PI).

✓ **When to Use STE**

- Some O/RM frameworks allow the use of POCO and STE. STE (*Self-Tracking Entities*) allow us to implement advanced aspects (like Optimistic Concurrency Management) with less effort, which is very useful and productive in N-Tier scenarios. So, **for N-Tier application scenarios and medium-size apps**, it is convenient to use STE because it offers a greater potential and less manual work to be done by us.

✓ **When to Use POCO**

- Pure POCO Domain entities is a more DDD oriented approach, as the POCO entities follows better the 'Persistence Ignorance principle'. Also, if we want our presentation layers to be developed/changed at a different pace from the Domain and Application layers, and we want internal changes in Domain entities not to impact presentation layers, it is better to use DTOs in conjunction with POCO entities. DTOs are specifically created for distributed services and used in the presentation layers and POCO Domain entities will be used only within the application server layers.
- Therefore, it is recommended to use POCO domain entities, which offer full independence from the persistence layer (fulfilling the PI principle). The use of DTOs is purer DDD (thanks to the decoupling between Domain entities and DTOs, which will ultimately be the presentation layer entities). However, it has a higher development cost and complexity due to data conversions required in both directions, from domain entities to DTOs and vice versa. This decision (STE versus DTO) is a design/architecture decision that greatly depends on the magnitude of the application. If there are several

development teams working on the same application, the DTOs decoupling entity models, will probably be more beneficial.

- The last option is a "mixed bag" which is, using STEs for N-Tier applications (to be used by the presentation layer, etc.) and simultaneously having a SOA layer specially designed for external integration and interoperability. Therefore, this SOA layer will be offered to other applications/external services that use web-service integration with DTOs, which is more simplified.



References

- *'ENTITY pattern'* in the book *'Domain Driven Design'* by Eric Evans.
 - **The Entity Design Pattern**
- <http://www.codeproject.com/KB/architecture/entitydesignpattern.aspx>



2.2.2.- Value-Object Pattern

“Many objects do not have conceptual identity. These objects describe certain characteristics of a thing.” [E.E.]

As we have seen before, tracking of the entity identity is crucial; however, there are many objects and data in a system that do not require such an identity and monitoring. In fact, in many cases this should not be done because it can impair the overall system performance in an aspect that, in many cases, is not necessary. Software design is an ongoing struggle with complexity, and if possible, this complexity should always be minimized. Therefore, we must make distinctions so that a special management is applied only when absolutely necessary.

The definition of VALUE-OBJECT is: *Objects that describe things*; to be more accurate, *an object with no conceptual identity that describes a domain aspect*. In short, these are objects that we instantiate to represent design elements which only concern us temporarily. We care about *what* they are, not *who* they are. Basic examples are numbers, strings, etc. but they also exist in higher level concepts. For example, an “Address” in a system could be an ENTITY because in that system an address is important as an identity. But in a different system, the “Address” can be simply a VALUE-OBJECT, a descriptive attribute of a company or person.

A VALUE-OBJECT can also be a set of other values or even references to other entities. For example, in an application that generates a Route to get from one point to another, that route would be a VALUE-OBJECT (it would be a “snapshot” of points to go through this route, but this route does not have an identity or the requirement to be persisted, etc.) even though internally it is referring to different Entities (Cities, Roads, etc.).

At the implementation level, a VALUE-OBJECT will normally have a short life without identity tracking.

From a different point of view, an ENTITY is usually composed of different attributes. For example, a person can be modeled as an Entity with an identity and composed internally by a set of attributes such as name, surname, address, etc., which are simply *Values*. From these values, those that are important to us as a set (like address) must be treated as VALUE-OBJECTS.

The following example shows a diagram of specific application classes where we emphasize what could be an ENTITY and what could subsequently be a VALUE-OBJECT within an ENTITY:

ENTITY vs. VALUE-OBJECT

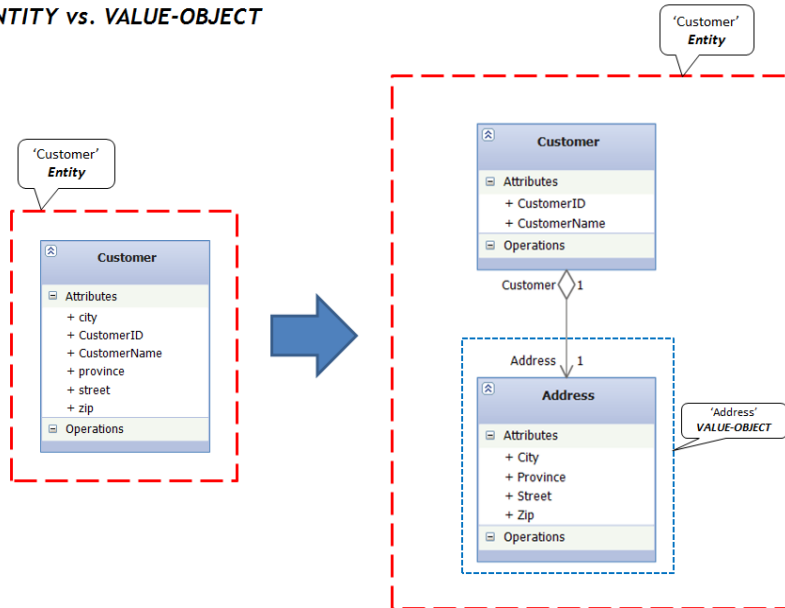


Figure 2.- Entities vs. Value-Object

Table 4.- Framework Architecture Guide Rules



Rule N°: D10.

Identifying and Implementing the VALUE-OBJECT Pattern when required

○ Recommendations

- When certain attributes of a model element are important to us as a group, but the object must not have a traceable identity, we must classify them as VALUE-OBJECTS. The meaning of these attributes should be expressed and they should have a related functionality. We must also treat the VALUE-OBJECT as immutable information throughout its life, from the moment it is created to the moment it is destroyed.



References

- **'VALUE-OBJECT' Pattern. By Martin Fowler.** Book 'Patterns of Enterprise Application Architecture': *"A small simple object, like money or a date range whose equality isn't based on identity."*
- **'VALUE-OBJECT' Pattern. Book 'Domain Driven Design' - Eric Evans.**

The attributes that comprise a VALUE-OBJECT should form a "conceptual whole". For example, street, city and zip code should not normally be separated into simple attributes within a Customer object (depending on the application domain, of course). In fact, those attributes are part of an 'Address VALUE-OBJECT', which simplifies the Customer entity object.

Design of VALUE-OBJECTS

Due to the lack of restrictions on VALUE-OBJECTS, these can be designed in different ways, but always favoring the most simplified design or what best optimizes the system performance. One of the restrictions of VALUE-OBJECTS is that their values must be immutable from their inception. Therefore, at their creation (construction) we must provide values and not allow them to change during the object lifetime.

Regarding performance, VALUE-OBJECTS allow us to perform certain "tricks", thanks to their immutable nature. This is especially true in systems where there may be thousands of VALUE-OBJECT instances with many coincidences of the same values. Their immutable nature would allow us to reuse them; they would be "interchangeable" objects, since their values are the same and they have no identity. This type of optimization can sometimes make a difference between software that runs slowly and another with good performance. Of course, all these recommendations depend on the application environment and deployment context. Sharing objects can sometimes provide better performance but in certain contexts (a distributed application, for example) it may be less scalable when having copies, because accessing a central point of shared reusable objects can cause a bottleneck in communications.



2.2.3.- AGGREGATE Pattern

An aggregate is a domain pattern used to define ownership and boundaries of the domain model objects.

A model can have any number of objects (entities and value-objects) and most of them will normally be related to many others. We will have, therefore, different types of associations. Most associations between objects must be reflected in the code and even in the database. For example, a one to one association between an employee and a company will be reflected as a reference between two objects and will probably imply a relationship between two database tables. If we talk about one to many relationships, the context is much more complicated. But there may be many relationships that are not essential to the particular Domain Model in which we are working. In short, it is hard to ensure consistency in changes of a model that has many complex associations.

Thus, **one of the goals we have to consider is to simplify the relationships in a domain entity model as much as possible.** This is where the **AGGREGATE pattern** appears. **An aggregate is a group/set of associated objects that are considered as a whole unit with regard to data changes.** The aggregate is delimited by a boundary that separates the internal objects from the external objects. Each aggregate has a root object (called root entity) and initially it will be the only accessible object from the outside (When performing logic operations). The root entity object has references to all of the objects that comprise the aggregate, but an external object can only have references to the root entity-object. If there are other entities or value-objects within the aggregate boundary, the identity of these entity-objects is only local and they only make sense if they belong to the aggregate. They do not make sense if they are isolated.

This single point of access to the aggregate (root entity) is precisely what ensures data integrity. Only aggregates can be queried directly from Repositories, anything else (persisted child entities and child value-objects) must be accessed from its related aggregate repository. From outside the aggregate, there is no access and no change of data to the aggregate secondary objects, only through the root, which implies a very important control level. If the root entity is erased, the rest of the aggregate objects must also be erased.

If the aggregate objects need to be persisted in the database, then they should only be accessed through the root entity. The secondary objects must be obtained through associations. **This implies that only the root entities of aggregates may have associated REPOSITORIES. The same happens at a higher level with the SERVICES. We can have SERVICES directly related to the AGGREGATE root entity, but never directly related to a secondary object of an aggregate.**

The internal objects of an aggregate, however, should be allowed to have references to root entities of other aggregates (or simple entities that do not belong to any complex aggregate).

In the following diagram we show an example of a possible AGGREGATE for a specific MODEL:

Aggregates (AGGREGATE pattern)

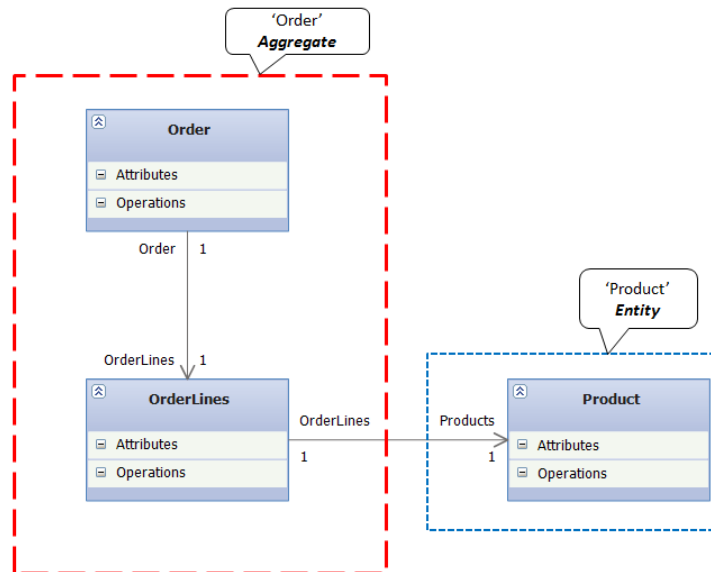


Figure 3.- Aggregates (AGGREGATE Pattern)

IMPORTANT

The former AGGREGATE is just an example of a possible aggregate. It doesn't have to be like that for every MODEL that contains similar entities.

AGGREGATES are not something you directly know when starting a project. It is not a matter of just aggregating entities and tables. **You won't be able to know what the aggregates of your MODEL are really, until you discover it while discussing with your Domain-Experts, talking about every single attribute of all the possible entities.**

We do really need to hear to the Domain Experts before defining our entities and aggregates. It is not a simple matter of creating an entity for each noun that the Domain Expert describes. Depending on their necessities, their use cases, we might need to create different entities, special entities for special relationships. Verbs and business tasks are even more important than the nouns they use, because many times a Domain Expert calls something with the same name/noun but may be it is for a different context. We first must understand the use cases, the business scenarios before defining any single entity.

Other than that, we'd be designing in a similar way we did in the old days with regular entity-relation diagrams, but now using entity-classes and aggregates. That is not the point.

Table 5.- Aggregates identification rule



Identify and Implement the AGGREGATE Pattern in the necessary cases to simplify the relationships between the model objects as much as possible

Recommendations

- **One of the objectives we have to consider is the ability to simplify the relationships in the domain entity model as much as possible. This is where the AGGREGATE pattern appears.** An aggregate is a group/set of associated objects that are considered as a whole unit with regard to data changes.
- **Keep in mind that this implies that only the aggregate root entities (or even simple entities) can have associated REPOSITORIES. The same happens on a higher level with SERVICES. We can have SERVICES directly related to the AGGREGATE root entity but never directly related to only a secondary object of an aggregate.**



References

- **'AGGREGATE' pattern.** Book '*Domain Driven Design*' - Eric Evans.



2.3.- Factory pattern

‘When creation of an object, or an entire AGGREGATE, becomes complicated or reveals too much of the internal structure, FACTORIES provide encapsulation. A program element whose responsibility is the creation of other object is called a FACTORY’. [E.E.]

If the creation of an object is not simple or we do need to make many validations and checks between several objects during their creation (like many AGGREGATES require), it is advisable to use FACTORIES in order to create our objects. On the other hand, if the creation of our object is really simple (like an Aggregate with only one entity), then, using its constructor and maybe relaying on a IoC/DI container would be enough for creating all its dependencies.

But the fact where we can use this pattern doesn’t mean that we have to use it always. Like mentioned, FACTORIES should only be used when they add value. If all they do is to wrap a new() operation, they don't add value.

Some scenarios in which factories add value are the following:

- When the creation of an object in itself is such a complex operation (i.e. in involves much more than just creating a new instance) that is best encapsulated behind an API (FACTORY). For instance, in our Sample-App related to this guide, we use FACTORIES for creating complex AGGEGATES (Like OrderFactory).
- Abstract Factories lets you vary Repository implementations independently of client code. This fits well with the 'L' in SOLID, but you could also achieve the same effect by using DI to inject the Repository into the Domain Service that requires it. Therefore, using a IoC/DI container, this scenario usually does not require the use of factories, as the IoC container will be acting as a factory creating all the Repository dependencies for us. It is in fact a better way, and it is the way we implemented our Sample-App related to this guide. Also, because this scenario is related to Repositories, it would be related to the Data-Persistence-Layer, instead the ‘Domain Model Layer’.

Complex object creation is a responsibility of the domain layer, yet this task does not belong to the objects that express the model (unless the task is very simple). There are Therefore, regarding the domain entities (and even complex Value-Objects), they start their lives in a Factory class (only when needed), that is, a DDD Factory, which means that the main purpose is to encapsulate the creation knowledge. What is created is a transient entity instance, with nothing being said about persistence as yet.



Rule N°: 1?

Use FACTORIES for Entities & Value-Objects creation when that creation process is complex.

Rule

If the creation of an object is not simple or we do need to make many validations and checks between several objects during their creation (like many AGGREGATES require), it is advisable to shift the responsibility for creating instances of those complex objects to a separate FACTORY, which may itself have no responsibility in the domain model but is still part of the domain design.

On the other hand, if the creation of our object is really simple (like an Aggregate with only one entity), then, using its constructor and maybe relaying on a IoC/DI container would be enough for creating all its dependencies (A POCO entity should not have external dependencies, therefore we wouldn't need an IoC container for creating simple POCO entities).



References

FACTORIES – (Pag. 136 – DDD Book, by Eric Evans)

Requirements for a good factory:

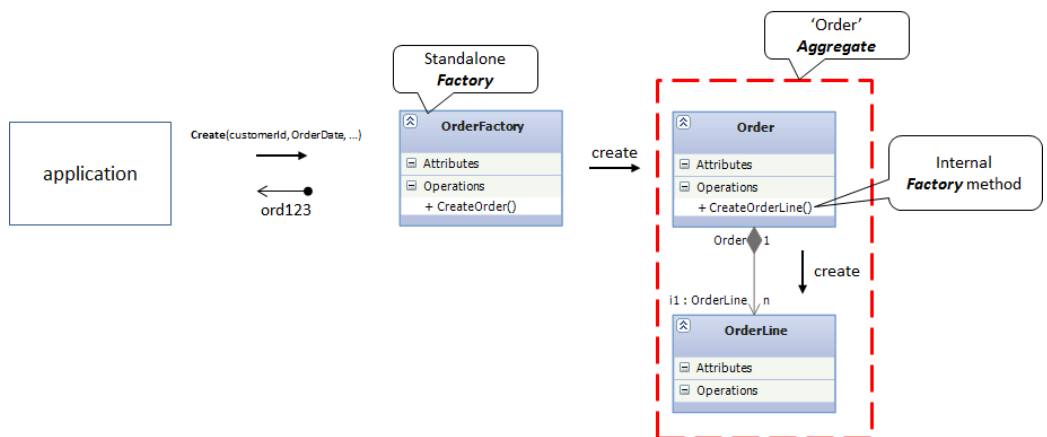
1.- Each creation method is atomic and enforces all invariants of the created object (AGGREGATE or VALUE-OBJECT). A factory should only be able to produce an object in a consistent state. Regarding ENTITIES, this means the creation of the entire AGGREGATE, with all invariants satisfied, but probably with optional elements to be added. For instance, an *OrderFactory* class should create the whole *Order* AGGREGATE, including the root entity *Order* and the child entity collection *OrderLines*. Then, in the coming future we will probably add orderliness to the AGGREGATE.

2.- Use the abstract type of the arguments (if available), not their concrete classes. The FACTORY is coupled to the concrete class it produces; it does not need to be coupled to concrete parameters also.

A standalone FACTORY usually produces an entire AGGREGATE (like the *Order* AGGREGATE), handing out a reference to the root entity (like the *Order* entity), and ensuring that the AGGREGATES's invariants are enforced. If an object interior to an AGGREGATE needs a factory, usually we should place the creation logic in a FACTORY METHOD on the root of the AGGREGATE. This hides the implementation of the interior of the AGGREGATE from any external client, while giving the root responsibility for ensuring the integrity of the AGGREGATE as elements are added. If the AGGREGATE root were not a reasonable home for the child entity factory, the go ahead and make a standalone FACTORY. But respect the rules limiting access within an AGGEGATE.

The following diagram shows an example of a factory related to a sample '**Order Aggregate**'.

Aggregate Factories



It is important to mention that **Factories can also be applied to build VALUE-OBJECTS** that have a complex creation, not only for ENTITIES and AGGREGATES. The concept is quite similar, but we have to remember that VALUE-OBJECTS are immutable, so this creation made by a factory will be the only moment where a VALUE-OBJECT is updated (or using a simple constructor if the creation process is not complex).



2.3.1.- Repository Contracts/Interfaces situated within the Domain Layer

The implementation of Repositories (Repository classes) is not part of the Domain but part of the infrastructure layers (since Repositories are linked to data persistence technology, such as an O/RM like *Entity Framework*); however, the contract referring to what such Repositories should look like (Interfaces to be implemented by these Repositories), should be a part of the Domain. That is why we include it here. The contract specifies what the Repository should offer, regardless of its internal implementation. These interfaces are agnostic to technology. Therefore, the Repository interfaces should be defined within the Domain layers. This point is highly recommended in DDD architectures and is based on the '*Separated Interface Pattern*' defined by Martin Fowler.

Logically, in order to comply with this pattern, Domain Entities and Value-Objects need to be **POCO**. In other words, they should be completely agnostic to the data access technologies. We must keep in mind that, in the end, domain entities are the actual "object types" of the parameters sent to and returned by the Repositories.

In conclusion, the goal of this design (*Persistence Ignorance*) is that the domain classes must know nothing at all about repositories' implementation. When we work in domain layers, we must ignore how the repositories are being implemented. It will also improve the code maintainability and readability.

Table 6.- Framework Architecture Guide rules



Rule N°: D12.

Define Repository interfaces within the Domain Layer following the SEPARATED INTERFACE PATTERN

○ Recommendations

- From the point of view of decoupling between the Domain Layer and the Data Access Infrastructure Layer, we recommend defining the Repository interfaces within the domain layer and the implementation of such interfaces within the Data persistence infrastructure layer. Thus, a Domain Model class may use a Repository interface as needed, without having to know the current Repository implementation, which has been implemented in the Infrastructure layer.
- This rule fits perfectly with decoupling techniques based on IoC containers.



References

- **‘Separated Interface’ pattern, by Martin Fowler.**
“Use Separated Interface to define an interface in one package but implement it in another. This way a client that needs the dependency to the interface can be completely unaware of the implementation.”
<http://www.martinfowler.com/eaCatalog/separatedInterface.html>



2.3.2.-Domain Model SERVICES

In most cases, our designs include operations that do not conceptually belong to Domain ENTITY objects. In these cases we can include/group such operations in explicit Domain Model SERVICES.

Note:

*It is important to point out that the SERVICE concept in a **DDD N-layered Architecture** is not a **DISTRIBUTED SERVICE** (typically Web Services) for remote access. A Web Service may “wrap” and publish the implementation of the Domain Service, but it is also possible for a Web application to have domain services and no Web Services.*

Those operations that do not specifically belong to Domain ENTITIES are inherently activities or operations, not internal characteristics of Domain Entities. But since our programming model is object oriented, we should also group them in objects. These objects are what we call SERVICES.

Forcing those Domain operations (in many cases, these are high level operations and group other actions) to be a part of the ENTITY objects would distort the definition of domain model and would make the ENTITIES appear artificial.

A SERVICE is an operation or set of operations offered simply as an artifact that is available in the model.

The word “Service” in SERVICE pattern precisely emphasizes what it offers: “What it can do and the actions it offers to the client that uses it, and highlights the relationship with other Domain objects (Covering several Entities in many cases).”

High level SERVICES (related to several entities) are usually called by activity names. In these cases, they are related to verbs associated with Use Case analysis, not to nouns, even when they may have an abstract definition of a Domain’s business

operation (for example, a Service-Transfer related to the action/verb “Transfer Money from one bank account to another.”)

The SERVICE operation names should come from the UBIQUITOUS LANGUAGE of the Domain. The parameters and return values should be Domain objects (ENTITIES or VALUE OBJECTS).

The SERVICE classes are also domain components, but in this case they are the highest level of objects within the Domain Layer. In most of the cases Domain Services cover different concepts and coordinate several related ENTITIES within business scenarios and use cases.

When a Domain operation is recognized as an important Domain concept, it should normally be included in a Domain SERVICE.

A Service should be stateless. This does not mean that the class implementing it should be static; it may well be an instantiable class (in fact, it has to be non-static if we want to use decoupling techniques between layers, such as IoC containers). A SERVICE being stateless means that a client program can use any instance of a service regardless of its internal state as an object.

Additionally, the execution of a SERVICE may use globally accessible information and may even change such global information (that is, it can have side effects). But the service should not have states that can affect its own behavior, like most domain objects do.

As for the type of rules to be included in the Domain SERVICES, a clear example would be in a banking application, making a transfer from one account to another, because it needs to coordinate business rules for an “Account” type with “Payment” and “Charge” type operations. In addition, the action/verb “Transfer” is a typical operation of a banking Domain. In this case, the SERVICE itself does not do a lot of work, it just coordinates the *Charge()* and *Pay()* method calls which are part of an Entity class, such as “BankAccount”. On the other hand, placing the *Transfer()* method in “Account” class could be a mistake (of course, this depends on the particular Domain) because the operation involves two “Accounts” and possibly other business rules to be considered.

Exceptions handling and business exceptions throwing should be implemented in both the Domain Services and the internal logic of entity classes.

From a perspective outside the domain, the SERVICES will usually be those that should be visible in order to perform relevant tasks/operations of each layer. In our example above (Bank Transfer), the SERVICE is precisely the backbone of the bank domain business rules.

Table 7.- Framework Architecture Guide



Design and Implement Domain SERVICES to Coordinate the Business Logic

○ Recommendations

- It is important to have these components in order to coordinate domain entities logic, and not to mix the domain logic (Business rules) with the application and data access logic coordination.
- A good SERVICE usually has this characteristic:
 - The operation is associated with a Domain concept that is not a natural part of an ENTITY internal logic.

References

- **SERVICE Pattern** - Book '*Domain Driven Design*' - **Eric Evans**.
- **SERVICE LAYER Pattern** – By **Martin Fowler**. Book '*Patterns of Enterprise Application Architecture*: “*Layer of services that establishes a set of available operations and coordinates the application response in each main operation.*”

Another rule to consider when dealing with the definition of data entities, and even classes and methods, is to define what we are really going to use. We should not define entities and methods because they seem logical, since in the end many of them will probably not be used in the application. In short, we should follow a useful recommendation from Agile-Methdologies called 'YAGNI' (*You Ain't Gonna Need It*), already mentioned at the beginning of this guide.

We should also define Domain Services only when we have to, like when there is really a need for **entity domain logic coordination**.

As shown in the following figure, we can have a domain service (in this case, the *BankTransferService* class) coordinating actions of the *BankAccount* entity business logic:

Relationship between Services and Domain Entities objects

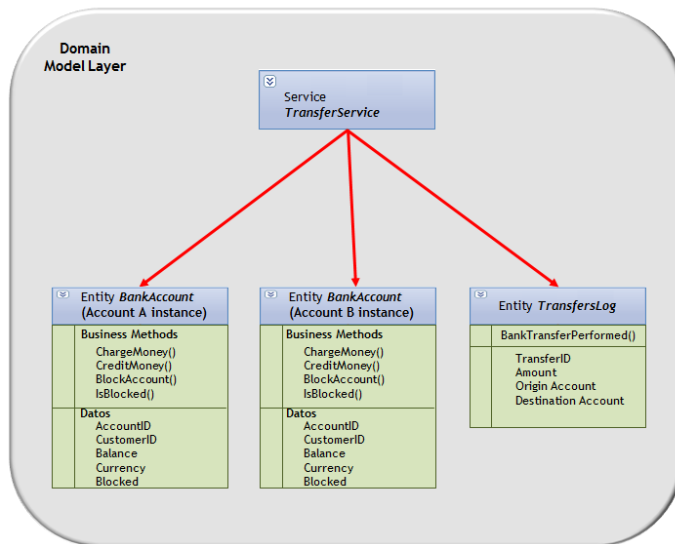


Figure 4.- Possible relationship between Entity and Service objects

A simplified UML sequence diagram (without considering transfer records) would have the following interactions. Basically, what we are pointing out is that calls between methods in this layer would be exclusively to execute the Domain logic whose flow or interaction could be discussed with a Domain expert or the end user:

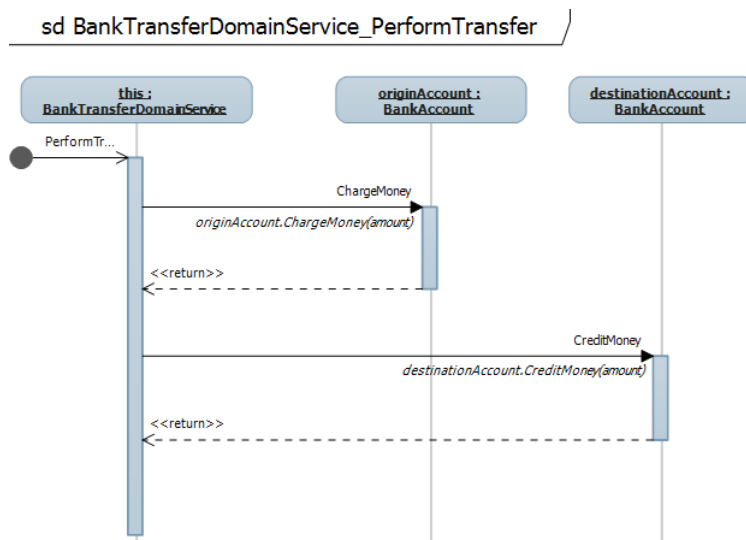




Figure 5.- Sequence Diagram for a simplified Bank transfer

Three objects appear in the sequence diagram. The first object (BankTransferDomainService) is a Domain Service that acts as the origin of the sequence and the other two (originAccount and destinationAccount, both instances of the BankAccount class) are “Domain Entity” objects, which would also have domain methods/logic (e.g. ChargeMoney and CreditMoney methods) that modify the “in memory data” of each domain entity object.


Table 8.- Domain Services should govern/coordinate the Business Logic

**Rule N°: D14.**

The Domain SERVICE classes must also govern/coordinate the Domain main processes

 **Rule**

- As a general rule, all complex business operations (**requiring more than a single operation unit**) related to different Domain Entities should be implemented within the ‘Domain SERVICE’ classes.
- Ultimately, this is about implementing the coordination of the whole use cases business logic.

**References**

SERVICE Pattern - ‘*Domain Driven Design*’ - Eric Evans.
SERVICE LAYER Pattern – By Martin Fowler. Book ‘*Patterns of Enterprise Application Architecture*’: “*Layer of services that establishes a set of available operations and coordinates the application response in each main operation.*”

Table 9.- Implementing only Domain Logic Coordination

**Rule N°: D15.**

Implementing only Domain logic coordination in the Domain Services

○ **Recommendation**

- Domain Service logic must be implemented with a very clean and clear code. Therefore, we must only implement the calls to the Domain lower level components (usually entity class logic), that is, only implement here the actions we would explain to a Domain/Business expert. Usually (with some exceptions), coordination of the application/infrastructure actions, such as calls to Repositories, creation of transactions, use of UoW objects, etc. should not be implemented here. These other actions for coordinating our application "plumbing" should be implemented within the Application Layer Services.
- This is a recommendation to make Domain classes much cleaner. However, mixing persistence coordination code, UoW and transactions with business logic code in Domain Services could be perfectly viable (many N-layered architectures, including DDD samples, do that). But, we do prefer a clear separation between the Domain Model Layer and Application Layer.
- Implement Domain Services only if they are necessary (YAGNI).



2.3.3.-SPECIFICATION Pattern

The **SPECIFICATION** pattern deals with **separating the decision as to which object types should be selected in a query from the object that makes the selection.** The Specification object will have a clear and limited responsibility that will be separated and decoupled from the Domain object that uses it.

This pattern is explained at the logical level and in detail in a paper written jointly by Martin Fowler and Eric Evans: <http://martinfowler.com/apsupp/spec.pdf>

Therefore, the main idea is for the decision of “what” candidate data must be retrieved to be separated from the candidate objects searched for, and from the mechanism used to retrieve them.

One of the nice benefits of the Specification pattern is that we can write code like the following:

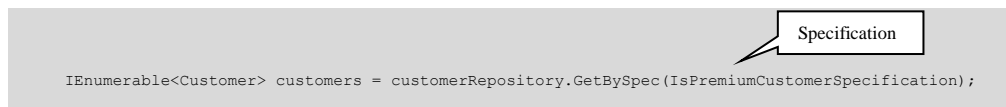


Figure 6.- Sequence Diagram for a simplified Bank transfer

Writing code like this allows reusing a specification from the domain within their related repository as a method for querying.

We will explain this pattern logically below, as originally defined by MF and EE. **However, in the Domain layer implementation section we will see that the implementation we chose differs from the original logical pattern due to a more powerful language offered by .NET. This refers specifically to expression trees, which provide better implementation if we only work with specifications for objects in memory, as described by MF and EE. However, we thought that it would be convenient to explain this here according to the original definition for a full understanding of the essence of this pattern.**

Cases in which the SPECIFICATION pattern is very useful

SPECIFICATIONS are particularly useful in applications where users are allowed to make open and compound queries, and “save” such types of queries in order to have them available in the future (e.g., a client analyst saves a compound query that he made which shows only customers of a certain country that have placed orders above \$200 plus other conditions that he has selected, etc.).

Subsumption Pattern (Related Pattern)

Once we use the **SPECIFICATION** pattern, another very useful pattern is the **SUBSUMPTION** pattern. Subsumption refers to the action and effect of subsuming. It comes from the prefix sub- and the Latin '*sumĕre*', which means “to take”; to include something, such as a component, in a comprehensive summary or classification or to consider something as a part of a broader package or as a special case subject to a general principle or rule (*Dictionary of the Royal Academy of the Spanish Language*).

In other words, the normal use of specifications tests these against a candidate object to see if this object meets all the requirements specified in the specification. Subsumption allows us to compare specifications to see if meeting one specification implies meeting a second one. Sometimes it is also possible to use the Subsumption pattern to implement this compliance. If a candidate object can produce a specification that characterizes it, then testing a specification is like a comparison of similar specifications. The Subsumption works especially well in Composite applications (Composite-Apps).


Since this logical concept of SUBSUMPTION starts to make things quite complicated for us, it is better to see the clarifying table offered by *Martin Fowler and Eric Evans* in their public '*paper*' on which pattern to use and how to use it depending on our needs:

Table 10.- SPECIFICATION Pattern Table – By MF and EE

Problems	Solution	Pattern
We need to select a subset of objects based on some criteria. We need to check that only certain objects are used for certain roles. We need to describe what an object can do without explaining the details of how the object does it and describe how a candidate could be built to meet the requirement.	Create a specification that is able to tell if a candidate object matches some criteria. The specification has a method <code>IsSatisfiedBy(anObject)</code> : Boolean that returns "true" if all criteria are met by the <code>anObject</code> .	SPECIFICATION
How do we implement a SPECIFICATION ?	We code the selection criteria into the <code>IsSatisfiedBy()</code> method as a block of code. We create attributes in the specification for values that commonly vary. We code the	Hard Coded SPECIFICATION Parameterized SPECIFICATION

	<p>IsSatisfiedBy() method to combine these parameters to make the test.</p> <p>Creating “leaf” elements for the various kinds of tests.</p> <p>Creating composite nodes for the ‘and’, ‘or’ and ‘not’ operators (see Combining Specifications below).</p>	COMPOSITE SPECIFICATIONS
How do we compare two specifications to see if one is a special case of the other, or is substitutable for another?	<p>Creating an operation called IsGeneralizationOf(Specification) that will answer whether the receiver is in every way equal or more general than the argument.</p>	SUBSUMPTION
<p>We need to figure out what still must be done to satisfy the requirements.</p> <p>We need to explain to the user why the Specification was not satisfied.</p>	<p>Adding a method RemainderUnsatisfiedBy() that returns a Specification that expresses only the requirements not met by the target object. (Best used together with Composite Specification).</p>	PARTIALLY SATISFIED SPECIFICATION

Table 11.- When to use the SPECIFICATION pattern



Rule N°: D16.

Use the SPECIFICATION pattern when designing and implementing dynamic or composite queries

Rule

- Identify parts of the application where this pattern is useful and use it when designing and implementing Domain components (creating Specifications) and implement specification execution within Repositories.

When to use the SPECIFICATION pattern

PROBLEM

- *Selection:* We need to select a set of objects based on certain criteria and

“refresh” the results in the application at certain time intervals.

- *Validation*: We need to ensure that only the proper objects are used for a particular purpose.
- *Construction to be requested*: We need to describe what an object could do without explaining the details on how it does it, but in a way that a candidate can be constructed to meet the requirement.

- **SOLUTION**

- Create a specification capable of telling if a candidate object meets certain criteria. The specification will have a Boolean `IsSatisfiedBy (anObject)` method that returns `True` if all criteria are met by that object.



Advantages of using Specifications

- We decouple the design of requirements, compliance and validation.
- It allows definition of clear and declarative queries.



When not to use the Specification Pattern

- We can fall into the anti-pattern of overusing the SPECIFICATION pattern and end up using it too much and for all types of objects. If we find out that we are not using the common methods of the SPECIFICATION pattern or that our specification object is actually representing a domain entity instead of placing restrictions on others, then we should reconsider the use of this pattern.
- In any case, we should not use it for all types of queries, only for those that we identify to be suitable for this pattern. We should not overuse it.



References

- *Specifications Paper* by Martin Fowler and Eric Evans:
 - <http://martinfowler.com/psupp/spec.pdf>

The original definition of this pattern [M.F. and E. E.], shown in the following UML diagram, explains how objects and object sets must satisfy a specification.

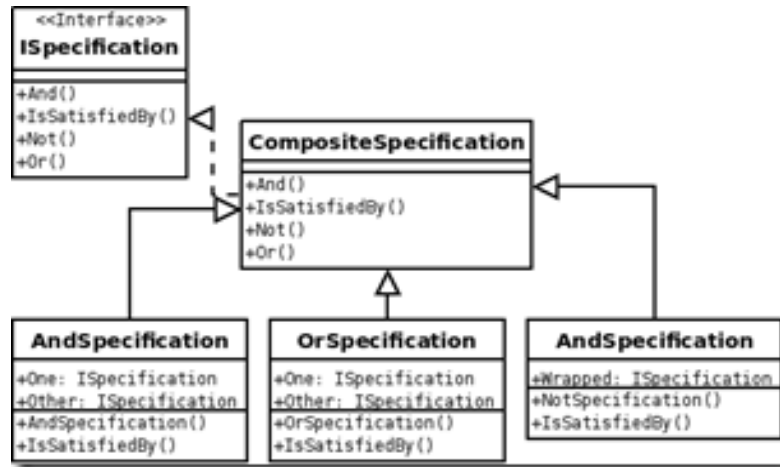


Figure 7.- UML diagram of the Specification pattern [by Martin Fowler and Eric Evans]

This is precisely what we stated as not making any sense in an advanced implementation when using .NET and EF (or other O/RMs) where we can work with queries that will be posed directly against the database instead of objects in memory, as the SPECIFICATION pattern originally suggested.

The main reason for the above statement comes from the pattern's own definition, which involves working with objects directly in memory; the **IsSatisfiedBy()** would take an instance of the object we want to check in order to determine if it meets certain criteria and return **true** or **false** depending on whether or not it is met. This, of course, is not desirable due to the overload entailed. Therefore, we could alter the definition of the SPECIFICATION slightly so that instead of returning a Boolean denying or confirming the compliance with a certain specification, it would return a statement with the criteria to be met.

This point will be covered and explained in more detail in our implementation of the SPECIFICATION pattern in the "Domain Layer Implementation" section, within this chapter.



2.4.- Domain Layer Design Considerations

When designing the Domain sub-layers, the main objective of the software architect should be to minimize the complexity, separating different tasks in different areas of concern and/or responsibility (for example, business processes, entities, etc., all of which represent different areas of responsibility.). The components we design for each area should target that specific area and should not include a code related to other areas of responsibility.

The following guidelines should be considered when designing the business layers:

- **Define Different Types of Domain Components:** It is always a good idea to have different types of objects that implement different types of patterns, according to the type of responsibility. This will improve the maintainability and reusability of the application code. For example, we can define domain **SERVICE** classes and other differentiated components for query **SPECIFICATION** contracts and, of course, Domain **ENTITY** classes. Finally, we can even have executions of workflow type business processes (a workflow with dynamic business rules, etc.), although we would normally be interested in placing the coordination of workflows at a higher level; in the Application layer and not in the Domain layer (It will depend of the nature of that workflow, domain rules vs. application rules).
- **Identify the Responsibilities of the Domain Layer:** The domain layer should be used to process business rules, transform data by domain logic requirements, apply policies, and implement validations related to the business requirements.
- **Design with High Cohesion.** The components should only contain specific functionality (concerns) associated with the component or entity.
- **Do not mix different types of components in the domain layers:** Domain layers should be used to decouple the business logic from the presentation and the data access code, and also to simplify unit testing of the business logic. Ultimately, this will dramatically increase the system's maintainability.
- **Reuse Common Business Logic:** It is good to use these business layers to centralize reusable business logic functions for different types of client applications (Web, RIA, Mobile, etc.).
- **Identify the Consumers of Domain Layers:** This will help determine how to expose the business layers. For example, if the presentation layer that will use the business layers is a traditional Web application, the best option is to access it directly. However, if the presentation layer is running on remote machines (RIA applications and/or *RichClient*), it will be necessary to render the Domain and Application layers through a Distributed Services layer (Web services).

- **Use abstractions to implement decoupled interfaces:** This can be achieved with interface type components, common definitions of interfaces or shared abstractions, where specific components depend on abstractions (interfaces) and not on other specific components. In other words, they do not directly depend on classes (this refers to the Dependency Injection principle for decoupling). This is especially important for the Domain SERVICES.
- **Avoid Circular Dependencies:** The business domain layers should only “know” the details related to the lower layers (Repository interfaces, etc.) and always (if possible), through abstractions (interfaces) and even through IoC containers. However, they should not directly “know” anything at all about the higher layers (e.g., Application layer, Service layer, Presentation layers, etc.).
- **Decoupling between the domain layers and the lower (Repositories) layers:** Additionally to the use of abstractions/intefaces, the most powerful techniques to achieve decoupling between internal layers are IoC/DI.



3.1.- Implementing Domain Entities

The first step we must take is to select a technology to implement the Domain entities. Entities are used to contain and manage our application main data and logic. In short, the domain entities are classes that contain values and render them through properties, but they also can and should render methods with business logic of the entity itself.

In the following sub-scheme we highlight where the entities are located within the Domain Model Layer:

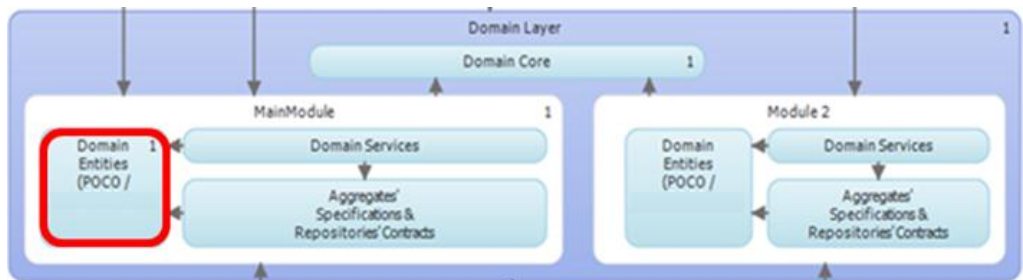


Figure 9.- Domain Entities situation

The decision of the data/technology type and format to be used for our domain entities is very important because it determines aspects affected by it, such as these questions:

- Is our Domain layer independent from the data access technology? Can we stop using our current data access technology and start using a different one while continuing to use our domain entity classes? The answer to this question would be totally different depending on the class type we are using for our domain entities (e.g., Datasets, custom classes, EF prescriptive classes, POCO, STE, etc.). For example, if we use DataSets as entities, then we certainly cannot simply change our data persistence framework, since that would require complete restructuring of the application, which will fully affect the heart of our application: the Domain Model Layer.
- In the event that we do not use DTOs and domain entities also travel to the presentation layer, the choice of entities type is even more critical as we are going to deal with interoperability issues and data serialization for Web Service remote communications, etc. Furthermore, the design and choice of technology to implement entities will greatly affect the performance and efficiency of the Domain layer.

Options of data/format/technology types:

- **POCO Classes**

As discussed above, our entities will be implemented as simple .NET classes with fields and properties for the entity attributes, and methods for the entity domain logic. The most important rule of POCO classes is that they should not have dependency on other components and/or classes from external frameworks. For example, a typical EF 1.0 entity is not POCO, since it depends on the EF entity base class. However, in EF 4.0 it is possible to use pure POCO classes that are completely independent from the EF stack. Because these **POCO classes are suitable for DDD N-layer architectures**, we will dig into this choice later on within this chapter.

In Entity Framework, POCO entities can be created manually from scratch (Since EF 4.1 *Code-First*) or with the help of a code generator which generates these classes automatically from an original entity EDMX data model (*Model First*). This feature is quite productive, but we need to have an existent 'Entity Data Model' before modeling our Domain Entity model (They are not exactly the same concept).

- **EF 4.0 Self-Tracking Entities**

EF 4.0 *Self-Tracking Entities (STE)* are simple objects that implement some interfaces required by the EF 4.0 'Self-Tracking' system. To be more specific, those interfaces are **IObjectWithChangeTracker** and **INotifyPropertyChanged**. The important point is that these interfaces do not belong to any particular persistence framework. **IObjectWithChangeTracker** is generated by the STE T4 templates and **INotifyPropertyChanged** is part of the standard .NET Framework library. Therefore, these interfaces do not depend on any particular framework except .NET.

The STE classes are very convenient for N-Tier architectures and medium applications where the client consumer should always be based on .NET or Silverlight in order to support entity-change-tracking and optimistic concurrency.

- **DataSets and DataTables (basic ADO.NET classes)**

The DataSets are similar to the disconnected databases in memory that are typically mapped fairly close to their own database schema. The use of DataSets is quite typical in .NET applications from the first version onwards, in a traditional and normal use of ADO.NET. The advantages of DataSets are that they are very easy to use, as well as being very productive in highly data oriented disconnected scenarios and CRUD applications (typically with an ADO.NET provider for a specific DBMS). "LINQ to DataSets" can also be used to work with them.

However, DataSets have important disadvantages, which should be seriously considered:

- 1.- DataSets are not interoperable toward other non-Microsoft platforms, such as Java. Even though they can be serialized to XML, they may cause problems if they are used as data types in Web services.

- 2.- Even in cases where interoperability with other platforms is not required, using them in web services is still not recommended, since DataSets are quite heavy objects, especially when serialized into XML in order to be used in Web services. The performance of our Web Services would be much higher if we use lighter POCO classes, STE classes and of course, custom DTOs. Therefore, we do not recommend the use of DataSets in communications across boundaries defined by Web services or even in inter-process communications (i.e. between different .exe processes).
- 3.- ORMs do not support/work with DataSets.
- 4.- DataSets are not designed to represent pure Domain entities with their domain logic included. The use of DataSets does not fit in a DDD Architecture because we will end up with an “Anemic Domain” having the domain entity logic separated (in parallel classes) from the domain entity data (in DataSets). Therefore, this option is not suitable for DDD architectural styles.

- **XML**

This is simply the use of fragments of XML text that contains structured data. Normally, we tend to make use of this option (representing domain entities with XML fragments) if the presentation layer requires XML or if the domain logic must work with XML content that should match specific XML schemes. Another advantage of XML is that, being simply formatted text, these entities will be fully interoperable.

For example, a system where this option would be commonly used is a message routing system where the logic routes the messages based on well-known nodes of the XML document. Keep in mind that the use and manipulation of XML may require great amounts of memory in scalable systems (many simultaneous users); if the XML volume is high, the access and process of the XML can also become a bottleneck when processing with standard APIs for XML documents.

The biggest problem with XML based entities is that it would not be “Domain Oriented” because we would have an “Anemic Domain” where the domain entity logic is separated from the domain entity data (XML). Therefore this option is not suitable in DDD either.

Table 12.- Domain Entities Rule



By default, Domain entities will be implemented as POCO classes or Self-Tracking Entities (STE)

Rule

- According to the considerations above, since this Framework Architecture is Domain Oriented and we should achieve maximum independence in the Domain objects, the domain entities will be implemented as POCO classes or STE. In EF 4.0. these are typically generated by T4 templates to save a lot of time in the implementation of these classes.
- Creating them manually (e.g., using a *Code-First* approach in .NET 4.1) is another viable option which is even a more "pure DDD" way, but it will take more work in order to handle 'Optimistic Concurrency', etc.



Advantages of POCO entities.

- They are independent of specific technology libraries.
- They are relatively lightweight classes that provide good performance.
- They are the most suitable option for DDD N-Layer Architectures.



When to use EF Self-Tracking Entities

- The use of **STEs** is recommended in most applications where we have complete control because they are more productive than POCO entities. STEs offer a very simplified optimistic concurrency management in N-Tier Applications. **The client application (presentation layer) needs to be .NET/Silverlight based.**
- **STEs are suitable for N-Tier applications where we control their end-to-end development. They are not, however, suitable for applications where there is no intention of sharing real data types between the client and server; for example, pure SOA applications where we only control one end, either the service or the consumer. In these cases, where there is no possibility or permission to share data types, it is recommended to make use of DTOs in distributed services (Web Services, etc.).**



When to use POCO Entities

- On the contrary, if our application is an application or service with a strong SOA orientation, then only DTOs should be used and we would be managing the concurrency aspects (Optimistic Concurrency managed by us, etc.). The use of **POCO** domain entities is recommended in these cases. The POCO option will result in some very simplified entities, although we will have to make much greater efforts in the implementation of our system (e.g., converting DTOs to domain entities, manual implementation of optimistic concurrency, etc.).



References

POCO in the Entity Framework: <http://blogs.msdn.com/adonet/archive/2009/05/21/poco-in-the-entity-framework-part-1-the-experience.aspx>

Self-Tracking Entities in the Entity Framework:

<http://blogs.msdn.com/efdesign/archive/2009/03/24/self-tracking-entities-in-the-entity-framework.aspx>



3.2.- Domain Entity options using Entity Framework

EF 4.1 (and future versions) provides the following options regarding what kind of entities' implementation we want to use:

- If following '**Model First**' or '**Database First**' approaches:
 - o *Prescriptive Entities* (coupled to EF base classes and *EntityObject-based* template)
 - Need to use partial classes to add entity logic
 - o *Self-Tracking Entities* (STE T4 Template and EF 4.0 *ObjectContext*)
 - Connected environment based on *ObjectContext*.
 - Need to use partial classes to add entity logic
 - o *POCO Entities* (POCO T4 Template and EF 4.0 *ObjectContext*)
 - Connected environment based on *ObjectContext*.
 - Need to use partial classes to add entity logic
 - o *POCO Entities* and EF 4.1 *DbContext* generator T4 Template
 - Connected environment based on *DbContext*.
 - Need to use partial classes to add entity logic
- If following a '**Code First**' approach:
 - o *POCO Entities* (Using your own POCO classes)
 - Connected environment based on *DbContext*.

- Directly mix data attributes and entity domain logic within your own POCO entity class.

If we want the purest DDD approach using .NET, the last option (***Code-First and Domain POCO entities plus DTOs for Distributed services***) is probably the most suitable, as it de-couples presentation layer data development (DTOs) from domain entities development (Domain). As handicap we have that all DTOs to Domain entities mapping, etc. will have to be handled/developed manually.

On the other hand, the *Self-Tracking Entities* approach is a balanced approach which provides more initial productive development because of its self-tracking data capabilities, but as long as the project gets larger, it would be less flexible than POCO-Code-First.

Please refer to the chapter on Distributed Services to analyze aspects regarding DTOs vs. STE for N-Tier applications.

In the first edition of this Architecture Guide we selected the STE choice (Code-First was not available at the time of the writing), but in this second edition we are going to focus on **POCO Entities and ‘Code First approach’** because it is a more DDD oriented choice.

Therefore, the selected way for the current implementation of our Domain Entities is ‘Code First’. ‘*Code First*’ is a new development pattern for the ADO.NET Entity Framework and provides an alternative to the existing *Database First* and *Model First* patterns. *Code First* is focused around defining your model using C#/Visual Basic .NET classes, these classes can then be mapped to an existing database or be used to generate a database schema. Additional configuration can be supplied using Data Annotations or via a fluent API.



3.3.- ‘Code First’ approach for implementing POCO Domain Entity classes

Especially for an applications with a long life and many Domain logic changes made by a development team other than the GUI team (Presentation Layer) or other consuming applications, as well as a very high number of entities (many hundreds of entities), the recommended choice would be the ‘***Code-First + DTOs approach***’ (DTOs especially if we are going to use Web Services for remote access to our server components).

Code First allows us to define our model using our own POCO C# or VB.Net classes. Additional optional configuration can be performed using attributes on our classes and properties or by using the *Fluent API*. Our model can be used to generate a database schema or to map to an existing database.

One of the advantages of *Code-First* towards DDD is that we can start focusing on our domain entities (the core of our Domain Model) as soon as possible without

thinking or implementing the database connection and mapping. This is why we changed the order of this chapter ‘Domain Model Layer’ which in this edition comes before the ‘*Infrastructure Data Persistence Layer*’ chapter. That is the way it should be in DDD: ‘*Let’s think first about our Domain Model*’, about our problem we want to solve, and we will map it later to any specific persistence technology (database). Using Code-First approach we can ‘play’ (define our Domain Entity Model and make some programming tests with it) with our POJO domain entity classes in an early stage, there’s no need to implement the database and its mapping until a later stage, when we are really sure we want those specific entities. We can even test our entities against a default database without digging in persistence details, thanks to ‘default conventions’ provided by ‘Code-First’ which we will explain later on.

The following figure illustrates a proposed way of working when modeling our application, starting on the Domain Model, not on Data Model.



Figure 10.- DDD oriented development process



3.4.- Setting up Entity Framework 4.1

Because EF 4.1 is newer than .NET 4.0 (it extends what we already had in EF 4.0), we need to install it from a regular

setup program (available from *Microsoft download Center*: <http://www.microsoft.com/downloads/en/details.aspx?FamilyID=b41c728e-9b4f-4331-a1a8-537d16c6acdf&displaylang=en>), or we can now install it using *NuGet*.

Note: *NuGet* is an *Open Source* project hosted at CodePlex (<http://nuget.codeplex.com>) which is in charge of administer and manage third party packages and libraries. If you don't know it, you can install it from Visual Studio 'Extension Manager'.

Once we have NuGet in Visual Studio, we just have to add a package called "Entity Framework", either from the context-menu in any project references or using NuGet commands console, typing '**Install-Package EntityFramework**' like we show below.

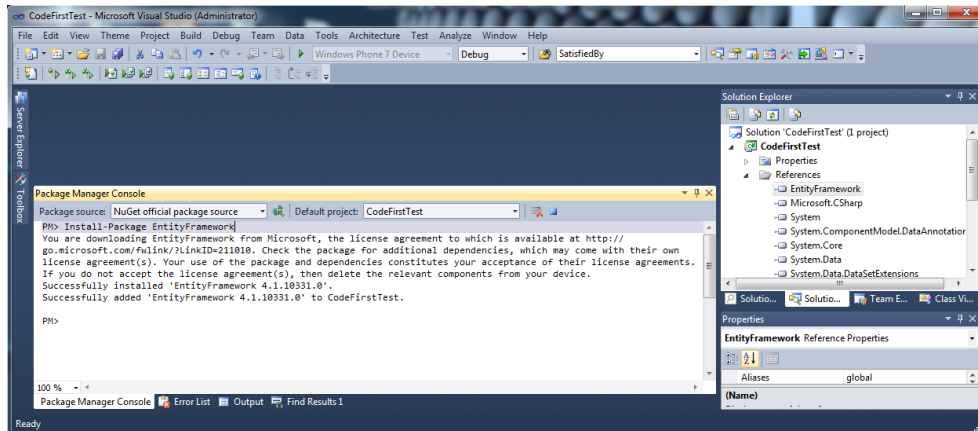


Figure 11.- Setting up EF 4.1

Once we add the 'Entity Framework package' we will have the reference to it already updated in our .NET project.



3.5.- Implementing Domain Entity Classes with ‘Code First’

At this time, we will be thinking in our Domain Model components, requirements, ubiquitous language, etc., and thanks to Code-first we can go on and directly define and implement our Domain entities as plain POCO classes within the Domain Model Layer. Therefore, we can even start ‘playing’ with our Domain Layer classes and we might have no ‘Infrastructure Data Persistence Layer’, yet. This is one of the reasons why Code-First fits better in DDD than ‘Database first’ or ‘Model First’ which are a bit more ‘Data/Database Driven’ approaches.

Within our ‘Domain Model Layer’, we would normally have an assembly (library class project) to store our Domain Entity classes.

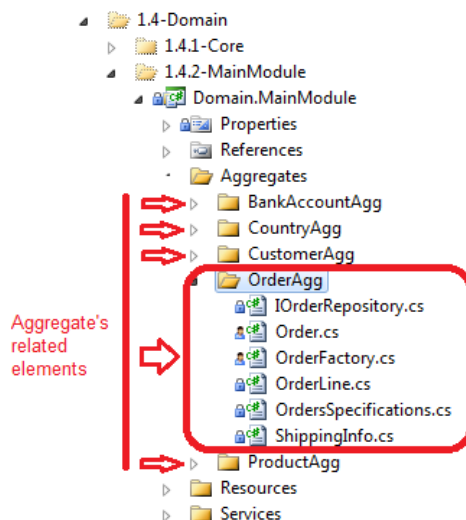


Figure 12.- Domain Aggregate Classes Structure Tree

Within that project, it is a good practice (related to DDD) to group entities within folders which represent AGGREGATES (like *OrderAgg*, *CustomerAgg*, etc.). Then, we can directly add POJO entity classes like the following two classes which would implement the ‘*Order Aggregate*’:

```
//POCO Entity Class (Root Entity Class for Order-Aggregate)

public class Order : Entity, IValidatableObject
{
    //Order lines set
    HashSet<OrderLine> _Lines;

    //Order info
    public DateTime OrderDate { get; set; }
    public DateTime? DeliveryDate { get; set; }
    public bool IsDelivered { get; set; }
}
```

Order: POJO Root Entity Class (For ‘Aggregate Order’)

```

public Guid CustomerId { get; set; }
public virtual Customer Customer { get; private set; }
public virtual ShippingInfo ShippingInformation { get; set; }

//...
//Omitted other Order entity methods
//...

public void SetOrderAsDelivered()
{
    this.DeliveryDate = DateTime.UtcNow;
    this.IsDelivered = true;
}

public decimal GetOrderTotal()
{
    decimal total = 0M;

    if (OrderLines != null //use OrderLines for lazy loading
        &&
        OrderLines.Any())
    {
        total = OrderLines.Aggregate(total,
                                     (t, l) => t += l.TotalLine);
    }

    return total;
}

public bool IsCreditValidForOrder()
{
    //Check if amount of order is valid for the customer credit
    decimal customerCredit = this.Customer.CreditLimit;

    if (this.GetOrderTotal() > customerCredit)
        return false;

    //TODO: This is a parametrizable value, you can
    //set this value in configuration or other system

    decimal maxTotalOrder = 1000000M;

    //Check if total order exceeds limits
    if (this.GetOrderTotal() > maxTotalOrder)
        return false;

    return true;
}

public virtual ICollection<OrderLine> OrderLines
{
    get
    {
        if (_Lines == null)
            _Lines = new HashSet<OrderLine>();

        return _Lines;
    }
    set
    {
        _Lines = new HashSet<OrderLine>(value);
    }
}

```

Method with Entity Logic

More Entity Logic

```
}
```

Figure 13.- Order Entity Root Class

```
//POCO Entity Class
public class OrderLine
    : Entity, IValidatableObject
{
    public decimal UnitPrice { get; set; }
    public int Amount { get; set; }
    public decimal Discount { get; set; }
    public Guid OrderId { get; set; }
    public Guid ProductId { get; set; }
    public Product Product { get; private set; }

    public decimal TotalLine
    {
        get
        {
            return (UnitPrice * Amount) * (1 - (Discount/100M));
        }
    }

    public void SetProduct(Product product)
    {
        if (product == null
            ||
            product.IsTransient())
        {
            throw new ArgumentNullException(Messages.
                exception_CannotAssociateTransientOrNullProduct);
        }

        //fix identifiers
        this.ProductId = product.Id;
        this.Product = product;
    }

    //Ommited other methods
    //...
}
```

Figure 14.- OrderLine - Child Entity Class

Using these plain classes and without needing any traditional EF 4.0 EDMX files (CSDL, SSDL, and MSL spaces for mapping), we are done. Using EF 4.1 and CODE-FIRST, we can directly create our domain entity model using classes, no steps needed regarding EDMX files.

Once we have the entity model, we could test its classes in memory or even going further, in a very few steps, we could persist it into a database. But we will do that in the next chapter which is dedicated to the '*Infrastructure Data Persistence Layer*'.



3.6.- Domain Logic in Entity Classes (No Anemic Domain)

In DDD it is essential to locate the logic (behavior) related to internal operations of an entity within the class of that entity itself. If the entity classes were used only as data structures and the entire domain logic was separated and placed into

the Domain Services, this would constitute an anti-pattern called “*Anemic Domain Model*” (See *Anemic Domain Model*, [Martin Fowler], <http://www.martinfowler.com/bliki/AnemicDomainModel.html>).

This anti-pattern explains that for Domain Oriented applications, it is not good to have an entity model containing ‘only data’, entities with only ‘gets & sets’ and no behavior related. This way is quite typical for people who were used to work with ADO.NET Datasets, VB ADO Recordsets or even EF V1.0. But, if we review all the business logic related to entities, we will discover that most Domain Entities have its own internal behavior. Sometimes it is just small methods validating data, other times are business methods performing calculations (like the method **GetNumberOfItems()** we put within the Order entity class) and many times it is critical business/domain logic which is really an intrinsic behavior of the entity. **If we don’t add behavior to entity classes, we will have to spread that behavior within Domain Services, and it will be a bad design decision because domain logic will be harder to maintain in a consistent state and business rules re-use will be more difficult. In short, what we want to do is to provide ‘object oriented sense’ to our entities, like Martin Fowler explained when talking about ‘anemic models’.**

Therefore, we should add the domain/business logic related to the internal part of each entity data within each entity class. If we use **Code-First** POCO Entity classes, the good news is that we can add that domain logic directly within the entity class itself. If we were using STEs or POCOs generated by T4 templates, we should have to add domain logic in ‘partial classes’. This is another reason why Code-First fits better with DDD.

For example, the following entity class *BankAccount* adds domain logic to the domain entity class itself. In particular, the operation being performed is “withdrawing/charging an account” and the business performs necessary checks before performing that operation to that bank account (Everything handled in-memory):

```
//POCO Domain Entity
//
//
public class BankAccount : Entity, IValidatableObject
{

    public class BankAccount
        :Entity, IValidatableObject
    {
        public string BankAccountNumber { get; set; }
        public decimal Balance { get; private set; }
        public bool Locked { get; private set; }
        public Guid CustomerId { get; set; }
        public virtual Customer Customer { get; private set; }

        //Ommitted BankAccountActivities
        //...

        public void Lock ()
        {
            if (!Locked)
                Locked = true;
        }
    }
}
```

POCO-Entity with Domain logic

Entity data properties


```

public void UnLock ()
{
    if (Locked)
        Locked = false;
}

public void WithdrawMoney (decimal amount, string reason)
{
    if ( amount < 0 ) throw new
    ArgumentException (Messages.exception_BankAccountInvalidWithdrawAmount);

    //WithdrawMoney is a term of our Ubiquitous Language.
    //Means deducting money to this account
    if (CanBeWithdrawed (amount))
    {
        checked
        {
            this.Balance -= amount;

            //anotate activity
            this.BankAccActivity.Add (new BankAccountActivity ()
            {
                Date = DateTime.UtcNow,
                Id = IdentityGenerator.NewSequentialGuid (),
                Amount = -amount,
                ActivityDescription = reason
            });
        }
    }
    else
        throw new
        InvalidOperationException (Messages.exception_BankAccountCannotWithdraw);
}

public bool CanBeWithdrawed (decimal amount)
{
    return !Locked && (this.Balance >= amount);
}
}

```

Domain Entity logic

Business checks / validations

Business/Domain logic for the BankAccount Withdraw process

Figure 15.- Entity Class with Domain Entity Logic

Many of these entity behaviors require us to change our property mapping, either because any property must not be public or other property must be readable but must not be directly updateable, etc.

For instance, we can extend our initial entities and add some other behavior. Because the OrderDetail is a child entity, it shouldn't be directly accessible. On the contrary, all operations against OrderDetail should be performed through the Aggregate Root entity, which in this case is the Order entity. Like the following:

```

//POCO Entity Class (Root Entity Class for Order-Aggregate)
public class Order : Entity, IValidatableObject
{
    //..Id property is inherited from the Entity base Class
    HashSet<OrderLine> _Lines;
    public Guid CustomerId { get; set; }
    public DateTime OrderDate { get; set; }
}

```

POCO Root Entity Class (For 'Aggregate Order')

```

public DateTime? DeliveryDate { get; set; }
public virtual ShippingInfo ShippingInformation { get; set; }

public void AddOrderLine(OrderLine line)
{
    if (line == null)
        throw new ArgumentNullException("line");

    //Fix relation
    line.OrderId = this.Id;

    this._Lines.Add(line);
}

//...
//More code omitted for brevity
//..
}

public class OrderLine : Entity, IValidatableObject
{
    public decimal UnitPrice { get; set; }
    public int Amount { get; set; }
    public decimal Discount { get; set; }

    public decimal TotalLine
    {
        get
        {
            return (UnitPrice * Amount) * (1 - Discount);
        }
    }

    public Guid OrderId { get; set; }
}

```

It is not allowed to add an Order-Line by any other way than this Entity-Root method with Domain logic

POCO Child Entity Class (For 'Aggregate Order')

Calculated & Read-Only property

Figure 15.- Root Entity Class with Domain Entity Logic

By the way, using EF STE and POCO T4 generated entities do not fit properly with this flexibility regarding mapping changes, hiding properties, etc. Using STE could be done within the T4 templates, but in many cases it is not so flexible. This is another reason why POCO-Code-First fits better with DDD.



3.7.- Implementing a Domain Entity Base class (Supertype pattern)

It is usually a recommended practice to have an Entity base-class so we can have common functionality which can be used by all of our entity classes. Typically, comparison methods or any other subject related to the IDENTITY of the entities is quite useful.

Below we expose an approach for a 'Entity Base Class' for POCO Code-First entities:

```
// Entity Base Class
//
public abstract class Entity
{
    int? _requestedHashCode;
    public Guid Id { get; set; }

    public bool IsTransient()
    {
        return this.Id == Guid.Empty;
    }

    public override bool Equals(object obj)
    {
        if (obj == null || !(obj is Entity))
            return false;

        Entity item = (Entity)obj;

        if (item.IsTransient() || this.IsTransient())
            return false;
        else
            return item.Id == this.Id;
    }

    public static bool operator ==(Entity left, Entity right)
    {
        return left.Equals(right);
    }

    public static bool operator !=(Entity left, Entity right)
    {
        return !left.Equals(right);
    }

    ...
    //Other methods
    ...
}
```

Base Class for all Domain Entities

Check if this entity is transient, ie, without identity at this moment

Entities comparison depend only on its IDENTITY (Id)

Operator ==

Operator !=

Figure 16.- Entity Base-Class

Therefore, we could now derive our domain entities so all that common entity behavior will be re-used. For instance, below we see the BankAccount entity deriving from the Entity base-class:

```
//POCO Domain Entity implementing our Entity Base-Class
//
public class BankAccount : Entity, IValidatableObject
{
    //Attributes
    //
    public string BankAccountNumber { get; set; }
    public decimal Balance { get; set; }
    public int CustomerId { get; set; }
    public bool Locked { get; set; }
    ...
    //
    //Other specific Domain Entity Logic (Methods)
    //
    ...
    ...
    ...
}
```

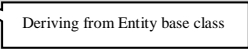


Figure 17.- BankAccount Domain Entity Class deriving from Entity base Class

The use of that base-class will be really useful especially regarding the *Identity* management as well as when comparing several instances using the operators we defined just once within the base class.



3.8.- Conventions in EF 4.1 'Code-first' Domain Entities

EF 4.1 'Code first' leverages a programming pattern referred to as convention over configuration. If you go ahead and use the Domain Entities we already stated in previous examples, and you use a simple EF DbContext you must create (this is related to database connection and infrastructure, so we will explain that in the next chapter), you could run your code and it would simply work. No need to create the database-tables or elaborate any kind of pre-established mapping.

```
// EF DbContext
// This code should be part of your Data Persistence Infrastructure Layer
public class MainBCUnitOfWork : DbContext, IMainBCUnitOfWork
{
    //SIMPLIFIED...
    public DbSet<Customer> Customers { get; set; }
    public DbSet<BankAccount> BankAccount { get; set; }
}
```




Figure 18.- Simple DbContext (Should be part of your Data Persistence Infrastructure Layer, not in Domain Model Layer)

DbContext is new in EF 4.1 and can be used from 'Code First approach' or you could also use it from 'Model/Database First' changing the T4 templates.

That is all the code we need to write to start storing and retrieving data. Obviously there is quite a bit going on behind the scenes and we will take a look at that in the following sections. Also, in our DDD Architecture we won't use a simple DbContext but we will extract a 'Unit of Work' interface and use Dependency Injection, etc.

What this means is that code first will assume that your classes follow the default conventions of the schema that EF uses for a conceptual model. In that case, EF will be able to work out the details it needs to do its job. The following code will simply work (It is an isolated proof of concept, not part of any sample).

```
// Using 'Code First'
class Program
{
    static void Main(string[] args)
    {
        using (var db = new MainBCUnitOfWork())
        {
            // Add a Customer
            var customer = new Customer {CustomerId = "ALFKI", Name = "Joe Smith"};
            db.Customers.Add(customer);
            int recordsAffected = db.SaveChanges();

            Console.WriteLine(
                "Saved {0} entities to the database, press any key to exit.",
                recordsAffected);

            Console.ReadKey();
        }
    }
}
```

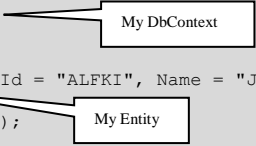


Figure 19.- Using 'Code-First' based on 'default conventions'

DbContext by convention can create a database for you on localhost\SQLEXPRESS. The database is named after the fully qualified name of your derived context. This is related to 'Data Persistence and Infrastructure', so we'll look at ways to change this later in the chapter dedicated to data persistence. Additionally, conventions work regarding 'Model Discovery'. DbContext worked out what classes to include in the model (it will translate it to database tables) by looking at the DbSet properties that we defined. It then uses the default Code First conventions to find primary keys, foreign keys etc.

For instance, DbContext inferred the entities and tables primary keys based on the attributes which have the post-fix 'Id'. Therefore, for our BankAccount entity, the primary key would be BankAccountId, by convention (**IdKeyDiscoveryConvention**). If we want to have several or composed keys, we will see that there are several ways to do it.

A list of some interesting default **conventions** is the following:

Table 13.- Main EF 4.1 'Default Conventions'

Convention	Description
IdKeyDiscoveryConvention	Convention to detect primary key properties.
ComplexTypeDiscoveryConvention	Convention to configure a type as a complex type if it has no primary key, no mapped base type, and no navigation properties.
ForeignKeyAssociationMultiplicityConvention	Convention to distinguish between optional and required relationships based on CLR nullability of the foreign key property.
ManyToManyCascadeDeleteConvention	Convention to add a cascade delete to the join table from both tables involved in a many-to-many relationship.
NavigationPropertyNameForeignKeyDiscoveryConvention	Convention to discover foreign key properties whose names are a combination of the dependent navigation property name and the principal type primary key property name(s).
OneToManyCascadeDeleteConvention	Convention to enable cascade delete for any required relationships.
OneToOneConstraintIntroductionConvention	Convention to configure the primary key(s) of the dependent entity type as foreign key(s) in a one-to-one relationship.
PluralizingEntitySetNameConvention	Convention to set the entity set name to be a pluralized version of the entity type name.
PluralizingTableNameConvention	Convention to set the table name to be a pluralized version of the entity type name.
PrimaryKeyNameForeignKeyDiscoveryConvention	Convention to discover foreign key properties whose names match the principal type primary key property name(s).
StoreGeneratedIdentityKeyConvention	Convention to configure integer primary keys to be identity.
PropertyMaxLengthConvention	Convention to set default maximum lengths for property types that support length facets.
TypeNameForeignKeyDiscoveryConvention	Convention to discover foreign key properties whose names are a combination of the principal type name and the principal type primary key property name(s).

There are many other conventions related to attributes we can use when customizing our model.

For a full list and explanation of all conventions in EF 4.1 see:
[http://msdn.microsoft.com/en-us/library/system.data.entity.modelconfiguration.conventions\(v=vs.103\).aspx](http://msdn.microsoft.com/en-us/library/system.data.entity.modelconfiguration.conventions(v=vs.103).aspx)

In case we do not want to apply any default convention, we are able to eliminate them, but this task is also part of our DbContext customization, therefore, we will explain it in the chapter dedicated to 'Data Persistence Infrastructure'.

However, if your classes do not follow those default conventions, you have the ability to add configurations to your classes to provide EF with the information it needs in order to match your classes with your existent database. This can be done through the use of two different ways: '**Data annotations**' or '**Fluent API**' (or both).



3.9.- 'Data Annotations' on Domain Entities

So far we have just let EF discover the model using its default conventions, but there are going to be times when our classes don't follow the conventions and we need to be able to perform further configuration. Like we stated, there are two options for this; we'll look at Data Annotations in this section and then the fluent API in the next section.

Suppose that the Id property in our BankAccount domain entity were not '**BankAccountId**' (using the 'Id' post-fix) but a property called '**BankAccountNumber**'. If we try to run our application we'd get an InvalidOperationException saying "*EntityType 'BankAccount' has no key defined. Define the key for this EntityType.*" because EF has no way of knowing that '**BankAccountNumber**' should be the primary key for the BankAccount entity.

To fix that, in this case we are going to use 'Data Annotations' so we need to add a reference:

Project -> Add Reference...

Select the ".NET" tab

Select "System.ComponentModel.DataAnnotations" from the list

Click "OK"

Add a using statement at the top of the .cs file:

```
using System.ComponentModel.DataAnnotations;
```

Take into account that this namespace is not part of EF but part of the .NET Framework itself, as 'Data Annotations' are also used in other technologies, not only in EF. If EF needs any other attribute which is not within that namespace, then you

will find it within the EntityFramework.dll, but then, this additional step will not be good for complying with the PI principle.

Now we can annotate the '**BankAccountNumber**' property to identify that it is the primary key:

```
//POCO Domain Entity using 'Data Annotations'
//
public class BankAccount : Entity
{
    //Attributes
    //
    [Key]
    public int BankAccountNumber { get; set; }

    public string BankAccountNumber { get; set; }
    public decimal Balance { get; set; }
    public int CustomerId { get; set; }
    public bool Locked { get; set; }
    //

    //Domain Entity Logic
    //
    ...
    ...
    ...
}
```

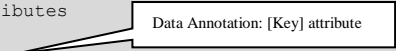


Figure 17.- BankAccount Domain Entity using 'Data annotations'

At this moment, we would be able to use the property/field **BankAccountNumber** as a primary key within our database.

Note: '*Data Annotations*' is not a new concept only available in EF. We can also use 'Data Annotations' in *ASP.NET Dynamic Data* and also in '*WCF RIA Services*'. In fact, these other technologies use the same assembly and namespace:

System.ComponentModel.DataAnnotations

There are many other attributes composing data annotations. Below are shown all the attributes classes for 'Data Annotations':

All Data Annotations attributes	Most commonly used attributes
<ul style="list-style-type: none"> System.ComponentModel.DataAnnotations System.ComponentModel.DataAnnotations.Resources AssociatedMetadataTypeTypeDescriptionProvider AssociatedMetadataTypeTypeDescriptor AssociationAttribute ColumnAttribute ComplexTypeAttribute ConcurrencyCheckAttribute CustomValidationAttribute DatabaseGeneratedAttribute DatabaseGeneratedOption DataType DataTypeAttribute DisplayAttribute DisplayColumnAttribute DisplayFormatAttribute EditableAttribute EnumDataTypeAttribute FilterUIHintAttribute ForeignKeyAttribute InversePropertyAttribute IValidatableObject KeyAttribute LocalizableString MaxLengthAttribute MetadataPropertyDescriptorWrapper MetadataTypeAttribute MinLengthAttribute NotMappedAttribute RangeAttribute RegularExpressionAttribute RequiredAttribute ScaffoldColumnAttribute ScaffoldTableAttribute StringLengthAttribute TableAttribute TimestampAttribute UIHintAttribute ValidationAttribute ValidationAttributeStore ValidationContext ValidationException ValidationResult Validator 	<ul style="list-style-type: none"> • KeyAttribute • StringLengthAttribute • MaxLengthAttribute • ConcurrencyCheckAttribute • RequiredAttribute • TimestampAttribute • ComplexTypeAttribute • ColumnAttribute • TableAttribute • InversePropertyAttribute • ForeignKeyAttribute • DatabaseGeneratedAttribute • NotMappedAttribute



3.10.- Entity Validations

Starting with EF 4.1, it is the first time that EF provides a mechanism for entity validations. This subject could seem quite simple, but it is not, and its implementation can have a big impact in most development projects where it has to be developed manually, from scratch.

It is important to differentiate the Domain Entity Model validations (regarding possible field values, etc.) from business/domain concepts validations (like when a field has to match certain regular expression). In general, an entity model validation can cause many unnecessary round-trips.

Entity Validations can be implemented in several ways, especially the following:

- *Data annotations (EF)*
- *Fluent API (EF)*
- Implementing *InvalidatableObject*

3.10.1.- Entity Validations using 'Data Annotations'

Like with any other subject, even though 'Data annotations' is very appealing and descriptive, it is also a more intrusive way on our entities, so in a DDD Entity Domain model we recommend not to use them unless those 'Data Annotations' are defined in the .NET Framework, within the *System.ComponentModel.DataAnnotations*. Other than that, we recommend to use your own *InvalidatableObject* implementation or **even the 'Fluent API' within the infrastructure layer**.

Below is shown a simplified entity example using 'Data annotations':

```
//Initial Customer Entity
//
public class Customer : Entity
{
    public int CustomerId { get; set; }
    [Required()]
    [MaxLength(20)]
    public string FirstName { get; set; }
    [Required()]
    [StringLength(20)]
    public string LastName { get; set; }
    public string City { get; set; }
    public string Street { get; set; }
    public string ZipCode { get; set; }
}
```

Using 'Data Annotation' attributes for **Validations**
(MaxLength is defined in EF assemblies !!!)

Using 'Data Annotation' attributes for **Validations**
(StringLength is defined in .NET assemblies)

Figure 27.- Initial Customer entity with validators

The problem with that code is when we are using an attribute which is defined in EF assemblies (like *MaxLength* which is defined in **EntityFramework.dll**), because in that case our entities won't be '*Persistence ignorant*' as they will be depending on EF assemblies. On the other hand, if we use attributes defined in .NET assemblies (like

StringLength which is defined within *System.ComponentModel.DataAnnotations*) then we have not a direct dependency between our entity and EF assemblies.

If we don't want to have EF or MVC dependencies in our validation attributes, we will have to check that we are using attributes deriving from *ValidationAttribute*. Therefore, we have the following available attributes:

Tabla 14.- System.ComponentModel.DataAnnotations validation annotations

Validation attribute	Purpose
StringLengthAttribute	Maximum length of characters of a string
RequiredAttribute	Element is required
RegularExpressionAttribute	Validation requires the value to match a specific regular expression
RangeAttribute	Validation that checks that the value is within a certain range
DataTypeAttribute	Specifies the name of an additional type to associate with a data field.
CustomValidationAttribute	Annotation that allows to link the validation with a custom validation

The attribute *CustomValidationAttribute* is special as it allows to delegate the validation to a custom validator (more complex, probably) that could be defined by us; therefore, it will be part of our Domain with no external dependency.

For instance, we could define a Custom-validator for a Credit-Card number, like shown below.

```
public static ValidationResult ValidateCCNumber(string creditCardNumber)
{
    bool result;

    //TODO: Validate DNI
    ... CC validation algorithm ...
    ...

    if (!result)
    {
        return new ValidationResult("Invalid CC number", new string[] { "CC" });
    }
    else
        return null;
}
```

Figure 28.- Custom Validator method

Then, we could decorate our entity like in the following code.

```
[CustomValidation(typeof(OrderValidation), "ValidateCCNumber")]
public string CreditCardNumber { get; set; }
```

Figure 29.- Applying a custom validator class and method

But, in general, the most decoupled way would be using *IValidatableObject* in the Domain Model Layer or 'Fluent API' validations implemented in the 'DataPersistence Infraestructura Layer'.

3.10.2.- Entity Validations using IValidatableObject

This way is quite clean regarding PI (Persistence Ignorant principle), as all of our validations code will be implemented within the domain model layer with no dependencies to external infrastructure types.

This way is especially indicated for logic validations where we also may need to validate entity states with some business rule that could even be related to several entities. Usually, this kind of validation can be seen as a 'Class level validation'.

In order to implement this kind of validation, the EF Product Group gave **IValidatableObject** support to POCO Code-First entities, so our entities can implement that interface to do more complex validations.

Something important regarding these validations is that they only will be processed if there are no errors at 'annotation level'

In the following code we show a possible implementation of an IValidatableObject validation.

```
public abstract class Product
    :Entity,IValidatableObject
{
    //...
    //Ommitted Product Entity properties and methods
    //...
    //...

    public IEnumerable<ValidationResult> Validate(ValidationContext
                                                validationContext)
    {
        var validationResults = new List<ValidationResult>();

        if (String.IsNullOrEmpty(Title) || String.IsNullOrWhiteSpace(Title))
            validationResults.Add(new ValidationResult(Messages.
                validation_ProductTitleCannotBeNullOrEmpty,
                new string[] { "Title" }));

        if (String.IsNullOrEmpty(Description) || String.
            IsNullOrEmpty(Description))
            validationResults.Add(new ValidationResult(
                Messages.validation_ProductDescriptionCannotBeNullOrEmpty,
                new string[] { "Description" }));

        if (AmountInStock < 0)
            validationResults.Add(new ValidationResult(Messages.
                validation_ProductAmountLessThanZero,
                new string[] { "AmountInStock" }));

        if (UnitPrice < 0)
            validationResults.Add(new ValidationResult(Messages.
                validation_ProductUnitPriceLessThanZero,
                new string[] { "UnitPrice" }));

        return validationResults;
    }
}
```





3.11.-‘Fluent API’ and the Domain Entities

We should make use of the Code First ‘Fluent API’ in order to change/customize our entity mappings, however, it must be written overriding **DbContext** methods (within our EF Context class). Specifically, we must override the ‘**DbContext OnModelCreating()**’ method. Therefore, and like we mentioned, that is part of the Data Persistence Infrastructure Layer, so we will explain how to use ‘Fluent API’ when dealing with that chapter.

IMPORTANT NOTE REGARDING DDD and PI Principle:

‘*Data Annotations*’ might initially look very appealing and easy to use when decorating our classes with attributes, but, on the other hand, and from a PI principle point of view, it is a bit intrusive on our Domain entities, as we need to write specific attributes that sometimes are linked to a specific persistence infrastructure, like EF 4.1 (Be aware of that).

On the other hand, using ‘**Default Conventions**’ or customizing it with ‘**Fluent API**’ is a way that fits much better with the PI (*Persistence Ignorant*), because our Domain Model will be ignorant of those mappings which will be defined into our Data Persistence Infrastructure layer, so, our Domain Model Layer will be better isolated from infrastructure implementation.

Therefore, if you are starting your project from scratch, the best way to go would probably be this: “*Base your entity classes on default conventions, as much as you can*”, because then your POCO domain entities will be cleaner, pure POCO!. Additionally, using ‘**Fluent API**’ or implementing your own ‘Validation system’ (implementing **IValidatableObject**), that won’t stain your POCO Domain Entities, either.



3.12.- Not Mapped Entity properties

Sometimes, when we are creating our domain entity model, we need entities containing several properties which should not be related/mapped to any database column.

When using Model/Database first (EDMX) and any T4 template code generator, we could use partial classes. Now we have a much elegant solution instead of creating another file for a partial class.

The attribute (Data annotation) '**NotMappedAttribute**' defined within the *System.ComponentModel.DataAnnotations* allows us to specify which properties we do not wish to map. Next code shows a property called 'TotalNumberItems' which is decorated with that attribute.

```
//Using 'NotMappedAttribute' Data Annotation
//
public class Customer : Entity
{
    //Omitted for brevity
    public string FirstName { get; set; }
    public string LastName { get; set; }

    [NotMapped()]
    public string FullName
    {
        get
        {
            return string.Format("{0}, {1}", this.LastName, this.FirstName);
        }
    }
}
```

'NotMappedAttribute' Data Annotation

Figure 18.- Customer Domain Entity using the 'NotMappedAttribute'

In case we want to use 'Fluent API' (less intrusive in our Domain entity classes), we can use the method 'Ignore()', as shown below.

```
//Using 'Ignore()' method Fluent API - (Code in Data Persistence Infr.Layer)
//
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>()
        .Ignore(c => c.FullName);
}
```

Ignore() method (Fluent API)

Figure 19.- Using 'Ignore()' method Fluent API

Take into account that when using 'Fluent API', the method 'OnModelCreating()' will be situated within the 'Data Persistence Infrastructure Layer', not in the Domain Model Layer which we are analyzing in current chapter.



3.13.- Read-Only Entity Properties

This feature can look like a very simple concept, but it is not so simple. When we are referring to a read-only property, that property value is calculated within the Domain Model, but cannot be 'Set' from the outside. Even though, that property is being stored in the database.

You could think that the following code would be enough in order to achieve a read-only property which is also being stored into the database.

```
//Wrong Code-First Read-Only property
//
public class Customer : Entity
{
    public int CustomerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public string FullName
    {
        get
        {
            return string.Format("{0}, {1}", this.LastName, this.FirstName);
        }
    }
}
```

Figure 20.- Wrong code for a Code-First Entity read-only property

But, that code is not right. It is, in fact, a read-only class-property, but it won't store that property into the database because it is not mapped. In order to make it work, **we only have to add an empty 'set' method**, like the following code.

```
//Using 'NotMappedAttribute' Data Annotation
//
public class Customer : Entity
{
    public int CustomerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public string FullName
    {
        get
        {
            return string.Format("{0}, {1}", this.LastName, this.FirstName);
        }
        set { }
    }
}
```

Empty set{} is needed

Figure 21.- Right code for a Code-First Entity read-only property

Doing so (it will be mapped), the read-only property will also be stored within the database.



3.14.- Creating POCO Proxies

If you want to **enable lazy loading for POCO entities** and have the Entity Framework **track changes** in your classes as the changes occur, your POCO classes must meet the requirements described in this topic so that the Entity Framework can create proxies for your POCO entities during run time. The proxy classes derive from your POCO types.

3.14.1.- POCO Entity Class Definition Requirements

The Entity Framework creates proxies for POCO entities if the classes meet the requirements described below. POCO entities can have proxy objects that support change tracking or lazy loading. You can have lazy loading proxies without meeting the requirements for change tracking proxies, but if you meet the change tracking proxy requirements, then the lazy loading proxy will be created as well. You can disable lazy loading by setting the *LazyLoadingEnabled* option to false.

For either of these proxies to be created:

- A **class** must be declared with **public access**.
- A **class** must **not** be **sealed**.
- A **class** must **not** be **abstract**.
- A **class** must have a public or protected constructor that does not have parameters.
- The **ProxyCreationEnabled** option must be set to true.

For lazy loading proxies:

- Each **navigation property** must have non-sealed, public, **and virtual** get accessor.

```
// Example of a Navigation Property ready for Proxy
public class BankAccount : Entity
{
    //... Ommitted ...
    public virtual ICollection<BankTransferLog> TransfersHistory { get; set; }
    //... Ommitted ...
}
```

'Virtual' in order to support POCO Proxy and Tracking

For change tracking proxies:

- Each **property** must have **non-sealed**, public, **and virtual** get and set accessors.
- A **navigation property** that represents the **"many" end of a relationship** must return a type that implements **System.Collections.Generic.ICollection of T**, where T is the type of the object at the other end of the relationship.
- If you want the proxy type to be created along with your object, use the **System.Data.Entity.DbSet.Create** method instead of the new operator when creating a new object.



3.15.- Complex-Types in 'Code-First'

Like Complex-Types are available in EF 4.0, they are also available when using EF 4.1 'Code-First'. It is really easy to implement. Let's suppose we have an initial *Customer* entity class with several properties like *City*, *Zipcode* and *Street*, and we realized that it would be better if we group those properties in a single complex-type called *Address*.

```
//Initial Customer Entity
//
public class Customer : Entity
{
    public int CustomerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string City { get; set; }
    public string Street { get; set; }
    public string ZipCode { get; set; }
}
```

Figure 22.- Initial plain Customer entity

```
//Initial Customer Entity
//
public class Customer : Entity
{
    public int CustomerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Address Address { get; set; }
}

public class Address
{
    public string City { get; set; }
    public string Street { get; set; }
    public string ZipCode { get; set; }
}
```

Address complex type

Figure 23.- Refactored Customer entity implementing a Complex-Type

Take into account that we don't need to do anything else, and the EF conventions will take care of our complex type and will map it to the database. The default convention for complex types set table columns representing those complex types as [ComplexTypeName]_[PropertyName]. Therefore, the internal SQL schema would be like the following.

```
CREATE TABLE [dbo].[Customers](
    [CustomerId] [int] IDENTITY(1,1) NOT NULL,
    [FirstName] [nvarchar](128) NULL,
    [LastName] [nvarchar](128) NULL,
    [Address_City] [nvarchar](128) NULL,
    [Address_ZipCode] [nvarchar](128) NULL,
    [Address_Street] [nvarchar](128) NULL,
    PRIMARY KEY CLUSTERED
    (
        [CustomerId] ASC
    )
)
```

```

)WITH (PAD INDEX = OFF, STATISTICS NORECOMPUTE = OFF,
IGNORE DUP KEY = OFF, ALLOW ROW LOCKS = ON, ALLOW PAGE LOCKS = ON)
ON [PRIMARY]
) ON [PRIMARY]

GO

```

Figure 24.- Default Table Schema when using a complex type

Again, like previously mentioned with regular entities, we can modify mapping parameters within the **'OnModelCreating()'** situated in the **'Data Persistence Infrastructure Layer' (Fluent API)**, or using 'Data Annotations'.

If we select the 'Fluent API' way, we will use the **ComplexType** method contained within the **DbModelBuilder** class, which let us change the mapping for a complex type, as the following code.

```

//Using 'ComplexType()' method Fluent API - (Code in Data Persistence Infr.Layer)
//
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>()
        .Ignore(c => c.FullName);
}

protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.ComplexType<Address>()
        .Property(p => p.City)
        .HasColumnName("City");

    modelBuilder.ComplexType<Address>()
        .Property(p => p.Street)
        .HasColumnName("Street");

    modelBuilder.ComplexType<Address>()
        .Property(p => p.ZipCode)
        .HasColumnName("ZipCode");
}

```

Using the **ComplexType** method (Fluent API)

Figure 25.- Using 'ComplexType()' method Fluent API

NOTE:

ComplexType<> does offer the same configuration possibilities than **Entity<>**.

We could also extract this mapping-customization putting that configuration into a new class (for instance called **AddressComplexTypeConfiguration**) which must derive from **ComplexTypeConfiguration<TComplexType>**, like the following code.

```

class AddressComplexTypeConfiguration
    : ComplexTypeConfiguration<Address>
{
    public AddressComplexTypeConfiguration()
    {
        this.Property(p => p.ZipCode)
            .HasColumnName("ZipCode");
    }
}

```

```

        this.Property(p => p.Street)
            .HasColumnName("Street");

        this.Property(p => p.City)
            .HasColumnName("City");
    }
}

```

Figure 25.- Creating a custom 'ComplexTypeConfiguration' classes

Then we need to add it to the configurations list within the `OnModelCreating()` method. This way is the recommended way for complex projects where we can have a lot of mapping configurations, so we can structure them instead of simply having a long configuration list within the `OnModelCreating()` method.

```

public class MainBCUnitOfWork : DbContext
{
    public IDbSet<Customer> Customers { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Configurations.Add(new AddressComplexTypeConfiguration());
    }
}

```

Adding our **ComplexType** configuration class

Figure 26.- Adding a custom 'ComplexTypeConfiguration' to the configurations

Again, this mapping code should be situated within the 'Data Persistence Infrastructure Layer'. We will explain the `OnModelCreating()` method in further details when dealing with the Data-Persistence Infrastructure Layer chapter.



3.16.-Implementing Value-Objects with .NET and Code-First

Like we already mentioned in the theory section of current chapter, a Value-Object has the following characteristics:

- Has no concept of an identity (Unlike Entities, where identity is crucial).
- Two different instances of a Value Object with the same values are considered equal.
- Must be immutable (Attributes values must be immutable).

A value object is therefore a set of attributes with some logic to manage that data.

A very simple .NET Value-Object could be like the following showed below. In this case it is a plain .NET Value-Object. It is still not compatible with EF Code-First, as it should also behave as a Complex-Type. Note how we are implementing its immutable characteristic. We can only initialize its attributes, after that, the attributes are not updateable from the outside:

```
//Sample .NET VALUE-OBJECT
public class Address
{
    private readonly string _city;
    private readonly string _zipCode;
    private readonly string _addressLine1;
    private readonly string _addressLine2;

    public Address(string city, string zipCode,
                  string addressLine1, string addressLine2)
    {
        _city = city;
        _zipCode = zipCode;
        _addressLine1 = addressLine1;
        _addressLine2 = addressLine2;
    }

    public string City
    {
        get { return _city; }
    }

    public string ZipCode
    {
        get { return _zipCode; }
    }

    public string AddressLine1
    {
        get { return _addressLine1; }
    }
    public string AddressLine2
    {
        get { return _addressLine2; }
    }
}
```

Plain .NET Value-Object (Not compatible with EF)

Values initialization only through the Constructor

Only 'get' for the properties (Immutable).

Figure 30.- Sample plain .NET 'Address Value-Object'

Unlike Entities, Value-Objects have no Identity. This is why we don't have any ID attribute.

The problem with that plain .NET value-object is that it won't work with EF Code-First approach. In order to make it work, we need an EF Complex Type (it has to have gets and sets), but at the same time we want it immutable!. So, the way to do that is putting **private sets** and of course, the constructor, like shown below.

```
//Sample VALUE-OBJECT Complex-Type  
public class Address
```

Figure 30.- Sample Complex-Type 'Address Value-Object'

```
//Sample VALUE-OBJECT Complex-Type  
public class Address  
    : ValueObject<Address>  
{  
    public string City { get; private set; }  
    public string ZipCode { get; private set; }  
    public string AddressLine1 { get; private set; }  
    public string AddressLine2 { get; private set; }  
  
    public Address(string city, string zipCode,  
                  string addressLine1, string addressLine2)  
    {  
        this.City = city;  
        this.ZipCode = zipCode;  
        this.AddressLine1 = addressLine1;  
        this.AddressLine2 = addressLine2;  
    }  
}
```

Figure 30.- Sample Complex-Type 'Address Value-Object'

So this value-object does work on EF 4.1 Code-First and it could be stored within a database.

But then, we have the other requirement: 'Two different instances of a Value Object with the same values are considered equal'. So we'd need to compare every attribute in order to make Value-Object instances comparisons. We could implement that comparison method in every Value-Object class, but for sure, it is a much better approach if we add that feature to a base class for Value-Objects, like we explain as the next section (This is why we are deriving from the **ValueObject<> class**).



3.17.- Implementing Entity/Aggregate Factories

We previously introduced the concept of the FACTORY pattern. In this section we are going to describe its implementation using .NET and showing an example of a specific Aggregate Factory.

3.17.1.- Standalone-Factories

A **standalone-factory** is nothing more than a class, and its purpose is to provide a method where it creates objects with a complex creation (usually standalone factories will create a whole AGGREGATE). A simplified aggregate factory can be the 'OrderFactory' shown below.

```
// Example of a Standalone-FACTORY for an AGGREGATE creation
public static class OrderFactory
{
    public static Order CreateOrder(Customer customer,
                                     string shippingName,
                                     string shippingCity,
                                     string shippingAddress,
                                     string shippingZipCode)
    {
        ShippingInfo shipping = new ShippingInfo(shippingName, shippingAddress,
                                                  shippingCity, shippingZipCode);

        Order order = new Order();
        order.OrderDate = DateTime.UtcNow;
        order.DeliveryDate = null;
        order.ShippingInformation = shipping;
        order.SetCustomer(customer);

        return order;
    }
}
```

Standalone FACTORY class

Specific invariant

3.17.2.- Factory-Methods

Then, within the ‘**Order**’ entity-root itself we could also add a FACTORY-METHOD called ‘CreateOrderLine()’ in order to create Order details with any associated logic.

```
// Example of a FACTORY-METHOD for a child ENTITY creation

public class Order : Entity, IValidatableObject
{
    HashSet<OrderLine> _Lines;
    public DateTime OrderDate { get; set; }
    public DateTime? DeliveryDate { get; set; }
    public bool IsDelivered { get; set; }
    //... Ommitted
    //...

    public virtual HashSet<OrderLine> OrderLines
    {
        get
        {
            if (_Lines == null)
                _Lines = new HashSet<OrderLine>();

            return Lines;
        }
        set
        {
            _Lines = value;
        }
    }

    public static OrderLine CreateOrderLine(Product product,
                                             int amount, decimal discount)
    {
        if (product == null
            || product.IsTransient())
        {
            throw new ArgumentException();
        }

        OrderLine orderLine = new OrderLine()
        {
            Amount = amount, //the # of items
            Discount = discount, //the associted discount
            UnitPrice = product.UnitPrice //set in this order line the current
price of the product
        };

        orderLine.SetProduct(product);

        return orderLine;
    }

    public void AddOrderLine(OrderLine line)
    {
        //... Ommitted
    }

    public decimal GetOrderTotal()
    {
        //... Ommitted
    }
}
```

Aggregate ENTITY-ROOT

FACTORY-METHOD (within an entity-root)



3.18.-Implementing Domain Services

In the following sub-scheme we emphasize where the “Domain SERVICES” classes are located within the Domain layer:



Figure 12.- Domain Services

As we already stated, a SERVICE is an operation or set of operations offered as an interface that is only available in the model.

The word “Service” comes from the SERVICE pattern and precisely emphasizes what is offered: *“What it can do and what actions are offered to the client by whom it is being used and emphasizes its relationship with other Domain objects (Incorporating several Entities, in some cases).”*

Normally, we will implement SERVICE classes as simple .NET classes with methods where the different possible actions related to one or several Domain entities are implemented; in short, implementation of coordination actions as methods.

Even though at the moment we only have defined the Entities and Value-Objects, but we have not review how we are going to persist/query against a Database, the Domain Services logic could still be defined and implemented as all the work we do within the Domain Services is applied against Entities & Value-Objects in-memory.

Therefore, the SERVICE classes should encapsulate and isolate the Domain Layer from any other layer like Application Layer and data persistence infrastructure layer.

It is in these Domain Services where most business rules coordination and calculations are performed (other than the logic which is intrinsic to the ENTITIES themselves) such as complex/global operations involving the use of multiple entity objects, as well as business data validations required for a whole process.



3.18.1.- Implementing Domain SERVICES as Business Process Coordinators

As explained in detail in the chapter on designing the Domain Layer Architecture, the **SERVICE** classes are primarily business process coordinators which normally cover different concepts and **ENTITIES** associated with scenarios and complete use cases.

A Domain service could also coordinate operations involving other related Domain Services and therefore, usually they will deal with several different Domain Entities.

This code is an example of a Domain SERVICE class aimed at coordinating a simplified Bank Transfer, where we are coordinating/invoking the BankAccount entity logic we already showed before:

C#

```
public class BankTransferService : IBankTransferService
{
    public void PerformTransfer(decimal amount,
                               BankAccount originAccount,
                               BankAccount destinationAccount)
    {
        // Check if customer has required credit
        // and if the BankAccount is not locked
        if (originAccount.CanBeWithdrawn(amount))
        {
            //Domain Logic
            //Process: Perform transfer operations to in-memory Domain-Model objects
            // 1.- Charge money to origin acc
            // 2.- Credit money to destination acc

            //Charge money
            originAccount.WithdrawMoney(amount,
                                       string.Format(Messages.
                                                       messages_TransactionFromMessage,
                                                       destinationAccount.Id));

            //Credit money
            destinationAccount.DepositMoney(amount,
                                           string.Format(Messages.
                                                           messages_TransactionToMessage,
                                                           originAccount.Id));
        }
        else
        {
            throw new InvalidOperationException(
                Messages.exception_BankAccountCannotWithdraw);
        }
    }
}
```

Diagram annotations:

- Domain Service**: Points to the `BankTransferService` class.
- Contract/Interface to be implemented**: Points to the `IBankTransferService` interface.
- Charge to Account**: Points to the `originAccount.WithdrawMoney` call.
- Pay to Account**: Points to the `destinationAccount.DepositMoney` call.

As it can be seen, the Domain Service code above is very clean and is only related to the business logic and business data. There are no “application plumbing”

operations such as the use of Repositories, Unit of work, transaction creation, etc. (Sometimes, however, we will need to use/call Repositories from within Domain Services when we need to query data depending on specific domain states).

In the Domain Service methods we simply interact with the logic offered by the entities that are involved. In the example above we call methods (*WithdrawMoney()*, *DepositMoney()*, etc.) that belong to the entities themselves.

Recall that, normally, in the implementation of Domain Service methods, all the operations are only performed against objects/entities in memory, and when the execution of our Domain Service method is completed, we have simply modified the data Entities and/or Value Objects of our domain model. Nonetheless, all these changes are still only in the server memory. The persistence of these objects and changes in data performed by our logic will not be applied until we coordinate/execute it from the ‘*Application Layer*’ which will invoke the Repositories within complex application logic (UoW and transactions).

The Application Layer, which is a higher layer level, will normally call Domain services, providing the necessary entities after having made the corresponding queries through the Repositories. And finally, this Application Layer will also be what coordinates the persistency in storages and databases. This application Layer will be explained in a following chapter.

Important:

Knowing how to answer the following question is important:

‘What code should I implement within the Domain Layer Services and Domain Entities?’

The answer is:

‘Only business operations we would discuss with a Domain Expert or an end user’.

We would not talk about “*application plumbing*”, how to create transactions, UoW, use of Repositories, persistence, etc. with a domain expert. On the other hand, the pure logic of the Domain should not be located in the Application Layer, so we do not get the Domain Logic “dirty”.

As mentioned above, the only exception to this is when a Domain Service needs to obtain and process data depending on specific and variable domain states. In that case, we will need to consume/call Repositories from Domain Services. This is usually done just to query data.



3.19.-Location of Repository Contracts/Interfaces in the Domain Layer

As explained during the theoretical chapters on DDD Architectural layers, the repository interfaces are the only things known about repositories by the Domain layer, and the instantiation of *Repository* classes will be made by the chosen IoC container (in this case *Unity*). Hence, we will have the Domain layer and the data persistence infrastructure layer's repository classes completely decoupled.

In the following sub-scheme we emphasized the location of the Repository contracts/interfaces within the Domain layer:



Figure 11.- Domain Contract Scheme

Thus, in our example, these interfaces will be defined in every Aggregate *namespace* within the Domain layer. For instance, a sample repository contract, like **ICustomerRepository** would be defined within the following namespace:

```
Microsoft.Samples.NLayerApp.Domain.MainBoundedContext.ERPModule.Aggregates.CustomerAgg
```

This would allow us to completely replace the data persistence infrastructure layer, the repositories themselves (the implementation) without affecting the Domain layer or having to change dependencies or making any re-compilations. Additionally, thanks to this decoupling, we can *mock* repositories and the domain business classes can dynamically instantiate “fake” classes (*stubs* or *mocks*) without having to change code or dependencies. This is done simply by specifying in the IoC container registration that, when it is asked to instantiate an object for a given interface, it must instantiate a fake (mock) class instead of the real one (both meeting the same interface, obviously).

Important:

Although the repository contract/interface should be located in the Domain layer for the reasons highlighted above, their implementation (*Repository class*) is really infrastructure code as it is linked to EF. Therefore, their implementation is explained with code examples in the chapter on “Implementing the Data Persistence Infrastructure Layer.”

Table 13.- Location of Repository Contract/Interface



Rule N°: 16

Locating Repository contracts/interfaces in the Domain Layer.

○ Rule

- To maximize the decoupling between the Domain Layer and the Data Access and Persistence Infrastructure Layer, it is important to locate the repository contracts/interfaces in the Domain Layer, and not in the Data Persistence layer itself.



References

Repository contracts in the Domain – (Book DDD by Eric Evans)

One example of Repository contract/interface within the Domain layer might be as follows:

C#

```
namespace Microsoft.Samples.NLayerApp.Domain.MainBoundedContext.  
ERPMModule.Aggregates.CustomerAgg  
{  
    //...  
    public interface ICustomerRepository : IRepository<Customer>  
    {  
        //Specific CustomerRepository method  
        IEnumerable<Customer> GetEnabled(int pageIndex, int pageCount);  
    }  
}
```

Repository Contract *Namespace* is within the Domain layer

It will inherit typical Repository methods

Specific Customer Repository method



3.20.-Location of Unit of Work Contract/Interface in the Domain Layer

The UoW is implemented within the persistence-infrastructure layer and mostly used from the application layer. But because of the same reason that we should locate Repository Contracts within the Domain Layer, we should also place a Unit of Work abstraction (interface) within the Domain Model Layer, that is, the contract to be complied by any UoW implementation in order to properly work with our Application and Domain Layers.

In fact, in order to comply with the **PI** (*Persistence Ignorance*) principle in a better way, we can define a UoW interface in the Domain Layer which would contain only the standard methods that the UoW pattern should have. Then, we can add another UoW interface in the Infrastructure-Persistence Layer which adds specific methods only available in EF context, which in our case is really the UoW implementation.

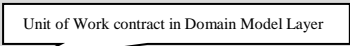
Therefore, we need to **create our UoW contract/interface**. We would better name its methods in a 'common language' (not tied to EF methods names), like the following interface.

```
C#
namespace Microsoft.Samples.NLayerApp.Domain.Core
//...Omitted
//...Omitted

public interface IUnitOfWork : IDisposable
{
    void Commit();

    void CommitAndRefreshChanges();

    void RollbackChanges();
}
...
...
}
```



In the next chapter (Data Persistence Infrastructure Layer) we will continue explaining the UoW implementation.



3.21.-QUERY SPECIFICATION Pattern

As explained in the Domain logic layer section, the **SPECIFICATION** pattern approach concerns **separating the decision on the object type to be selected in a query from the object that performs the selection**. The “Specification” object will have a clear and limited responsibility that should be separated and decoupled from the Domain object that uses it.

Therefore, the main idea is that the decision of “what” candidate data is to be obtained should be separated from the candidate objects searched for and the mechanism used to find them.

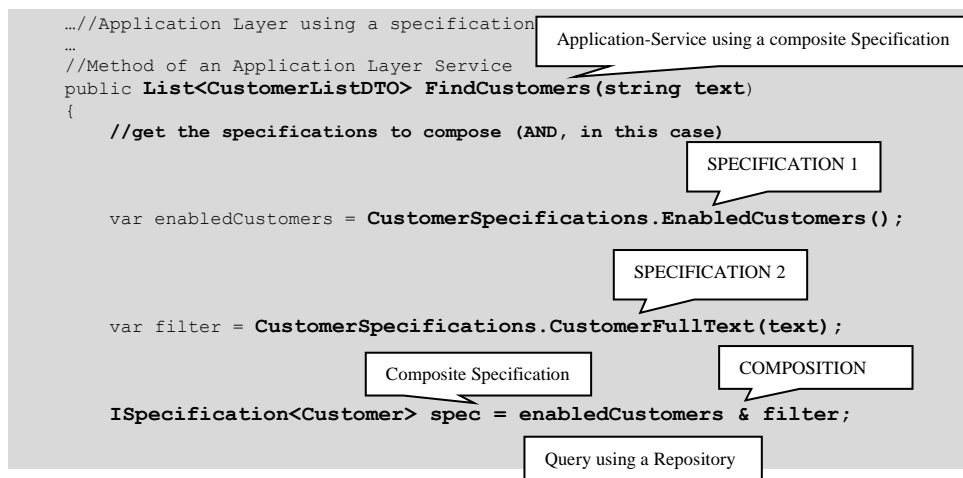
These tasks (query definitions) could be directly done using **Lambda expressions**, but then it is more complicated to make **COMPOSITE SPECIFICATIONS**. This is why we are implementing this pattern which, at last, is generating **Lambda expressions**, but constructed in a structured and composite way



3.21.1.- Use of the QUERY SPECIFICATION Pattern

Although specifications are **defined within the Domain Model Layer** because they are related to AGGREGATES, however, the *specification* pattern will usually be used/consumed from the **application layer** where we usually coordinate the logical queries we want to execute against the persistence-infrastructure layer. The goal is to decouple the actual implementation of these logical queries (specifications) from the data access and persistence infrastructure layers.

Just for understanding why we could want to have specifications, below we show the code where we **use/consume** several specification (Usually at the Application Layer).



```

//Query this criteria
var customers = _customerRepository.AllMatching(spec);

if (customers != null
    &&
    customers.Any())
{
    //return adapted data
    return _typesAdapter.Adapt<IEnumerable<Customer>,
List<CustomerListDTO>>(customers);
}
else // no data..
    return null;
}

```

3.21.2.- Implementation of the SPECIFICATION Pattern

Now, the implementation of SPECIFICATIONS does is on the Domain Model Layer.

The implementation we choose differs in part from the original logical pattern defined by MF and EE. This is due to the increased power of the language offered by .NET, such as expression trees, where we can achieve much greater benefit than if we work only with specifications for objects in memory, as originally defined by MF and EE.

Having the possibility of an advanced implementation with .NET and Linq, where we can work with queries that will be posed directly against the database will get a much better performance than working only with objects in memory, as originally stated the SPECIFICATION pattern (at that time, Linq and lambda expressions didn't exist).

The main reason for the statement above derives from the definition of the pattern. It involves working with objects directly in memory because the **IsSatisfiedBy()** method will take an instance of the object/s where we want to confirm if it meets certain criteria and return **true** or **false** depending on whether there is compliance or not. This, of course, is undesirable due to the overload it would entail. Given all of the above, we might modify our SPECIFICATION definition a bit so that, instead of returning a Boolean denying or confirming the compliance with a certain specification, **it returns an expression with the criteria to be met.**

In the following code fragment we would have a skeleton of our base contract with this slight modification:

```

C#

public interface ISpecification<TEntity>
    where TEntity : class
{
    /// <summary>
    /// Check if this specification is satisfied by a
    /// specific lambda expression
    /// </summary>
    /// <returns></returns>
    Expression<Func<TEntity, bool>> SatisfiedBy();
}

```

Skeleton/Interface of our base contract

We use SatisfiedBy() instead of the original IsSatisfiedBy()

At this point we could say that we already have the base and the idea of what we want to build, so now all we need to do is follow the rules and guidelines of this pattern to begin creating our direct specifications (“*hard coded specifications*”), as well as our composite specifications like “AND”, “OR”, etc.

Table 14.- Objective of SPECIFICATION pattern implementation

Objective of SPECIFICATION pattern implementation

While maintaining the principle of Responsibility Segregation, and considering that a QUERY SPECIFICATION is a business concept (a special search type, that is perfectly explicit) we are looking for an elegant manner to perform different queries in terms of parameters using expressions, conjunctions or disjunctions (Composition).

We could create query specifications as follows:

```
C#
public static class OrdersSpecifications
{
    public static ISpecification<Order> OrdersByCustomer(Customer customer)
    {
        if (customer == null
            ||
            customer.IsTransient())
        {
            throw new ArgumentNullException("customer");
        }

        return new DirectSpecification<Order>(o => o.CustomerId == customer.Id);
    }

    public static ISpecification<Order> OrderFromDateRange(DateTime? startDate,
                                                             DateTime? endDate)
    {
        Specification<Order> spec = new TrueSpecification<Order>();

        if (startDate.HasValue)
            spec &= new DirectSpecification<Order>(o => o.OrderDate >
                                                    (startDate ?? DateTime.MinValue));

        if (endDate.HasValue)
            spec &= new DirectSpecification<Order>(o => o.OrderDate <
                                                    (endDate ?? DateTime.MaxValue));

        return spec;
    }
}
```

Returns a Lambda Expression

Returns a Lambda Expression

IMPORTANT: Please note how the above specifications provide us with a mechanism to know the criteria of the elements we want to search for; but it does not know anything about who will perform the search operation thereof. In addition to this clear separation of concerns, the creation of these elements also helps us to make important domain operations, such as types of search criteria, perfectly clear. Otherwise, these would be scattered around different parts of the code, making them more difficult and expensive to modify. Finally, another advantage of specifications, as proposed here, is the possibility of performing logical operations on them, which provides us with a simple mechanism to perform dynamic queries in LINQ.



3.21.3.- Composing Specifications with Operators (AND/OR)

Although there is evidently more than one approach to implement these operators, we choose to implement them with the VISITOR pattern to evaluate the expressions (ExpressionVisitor:

[http://msdn.microsoft.com/en-us/library/system.linq.expressions.expressionvisitor\(v5.100\).aspx](http://msdn.microsoft.com/en-us/library/system.linq.expressions.expressionvisitor(v5.100).aspx)).

What we need is the following class to come up with a re-composition of the expression instead of an **InvocationExpression** (not compatible with EF).

```
C#
/// <summary>
/// Extension methods for add And and Or with parameters rebinder
/// </summary>
public static class ExpressionBuilder
{
    public static Expression<T> Compose<T>(this Expression<T> first,
        Expression<T> second,
        Func<Expression, Expression, Expression> merge)
    {
        // build parameter map (from parameters of second to parameters of first)
        var map = first.Parameters.Select((f, i) => new {f,s=second.Parameters[i]}).
            ToDictionary(p => p.s, p => p.f);

        // replace parameters in the second lambda expression
        // with parameters from the first lambda expression
        var secondBody = ParameterRebinder.ReplaceParameters(map, second.Body);

        // apply composition of lambda expression bodies
        // to parameters from the first expression
        return Expression.Lambda<T>(merge(first.Body,
            secondBody),
            first.Parameters);
    }

    public static Expression<Func<T, bool>> And<T>(
        this Expression<Func<T, bool>> first,
        Expression<Func<T, bool>> second
    )
    {
        return first.Compose(second, Expression.And);
    }

    public static Expression<Func<T, bool>> Or<T>(
        this Expression<Func<T, bool>> first,
        Expression<Func<T, bool>> second)
    {
        return first.Compose(second, Expression.Or);
    }
}
```

EXPRESSIONS builder

Compose two expressions and merge all in a new expression

AND operator

OR operator

The complete definition of an AND specification, therefore, is shown in the following code:

```
C#  
  
public sealed class AndSpecification<T>  
    : CompositeSpecification<T>  
    where T : class  
{  
    private ISpecification<T> RightSideSpecification = null;  
    private ISpecification<T> LeftSideSpecification = null;  
  
    public AndSpecification(ISpecification<T> leftSide,  
        ISpecification<T> rightSide)  
    {  
        if (leftSide == (ISpecification<T>)null)  
            throw new ArgumentNullException("leftSide");  
  
        if (rightSide == (ISpecification<T>)null)  
            throw new ArgumentNullException("rightSide");  
  
        this.LeftSideSpecification = leftSide;  
        this.RightSideSpecification = rightSide;  
    }  
  
    public override ISpecification<T> LeftSideSpecification  
    {  
        get { return LeftSideSpecification; }  
    }  
  
    public override ISpecification<T> RightSideSpecification  
    {  
        get { return RightSideSpecification; }  
    }  
  
    public override Expression<Func<T, bool>> SatisfiedBy()  
    {  
        Expression<Func<T, bool>> left =  
            _LeftSideSpecification.SatisfiedBy();  
        Expression<Func<T, bool>> right =  
            RightSideSpecification.SatisfiedBy();  
  
        return (left.And(right));  
    }  
}
```

AND specification

The SatisfiedBy() method required by our SPECIFICATION pattern

The hierarchy of specifications proposed in the paper by **Evans and Fowler** includes others specifications we can also build (OR, NOT, TRUE, etc.), like we did in our Sample-Application.



3.22.-Implementing Unit Testing for the Domain Layer

Like most elements of the solution, our Domain Layer is one of the areas that should be covered by unit testing. It should, of course, meet the same requirements demanded in the rest of the layers or parts of a project. The main items within this layer that must have good code coverage are the entities and the domain services.

Basically, we can have a Domain Layer Testing project added for each Bounded-Context. If we have a *MainBoundedContext* projects, we should have a test project such as *Domain.MainBoundedContext.Tests* where we will have a set of tests for entities, aggregate-factories, services, etc. (Although you can have any other Unit Testing approach, this is very personal, but in any case, you must do unit testing for your domain model layer).

In the following code example we can see some tests of the domain entity *BankAccount*.

```
C#
[TestClass()]
public class BankAccountAggTests
{
    [TestMethod()]
    public void BankAccountFactoryCreateValidBankAccount()
    {
        //Arrange
        Customer customer = CustomerFactory.CreateCustomer("John", "Smith",
IdentityGenerator.NewSequentialGuid(), new Address("city", "zipcode", "AddressLine1",
"AddressLine2"));
        customer.Id = IdentityGenerator.NewSequentialGuid();
        string bankAccountNumber = "AB001";
        BankAccount bankAccount = null;

        //Act
        bankAccount = BankAccountFactory.CreateBankAccount(customer,
bankAccountNumber);
        var validationContext = new ValidationContext(bankAccount, null, null);
        var validationResult = customer.Validate(validationContext);

        //Assert
        Assert.IsNotNull(bankAccount);
        Assert.IsTrue(bankAccount.BankAccountNumber == bankAccountNumber);
        Assert.IsFalse(bankAccount.Locked);
        Assert.IsTrue(bankAccount.CustomerId == customer.Id);

        Assert.IsFalse(validationResults.Any());
    }

    [TestMethod()]
    [ExpectedException(typeof(InvalidOperationException))]
    public void BankAccountCannotWithdrawMoneyInLockedAccount()
    {
        //Arrange

        Customer customer = CustomerFactory.CreateCustomer("John", "Smith",
IdentityGenerator.NewSequentialGuid(), new Address("city", "zipcode", "AddressLine1",
"AddressLine2"));
        customer.Id = IdentityGenerator.NewSequentialGuid();
```

```

        string bankAccountNumber = "AB001";
        BankAccount bankAccount = BankAccountFactory.CreateBankAccount(customer,
bankAccountNumber);
        bankAccount.Lock();

        //Act
        bankAccount.WithdrawMoney(200, "the reason..");
    }

    [TestMethod()]
    [ExpectedException(typeof(InvalidOperationException))]
    public void BankAccountCannotWithdrawMoneyIfBalanceIsLessThanAmountAccount()
    {
        //Arrange

        Customer customer = CustomerFactory.CreateCustomer("John", "Smith",
IdentityGenerator.NewSequentialGuid(), new Address("city", "zipcode", "AddressLine1",
"AddressLine2"));
        customer.Id = IdentityGenerator.NewSequentialGuid();

        string bankAccountNumber = "AB001";
        BankAccount bankAccount = BankAccountFactory.CreateBankAccount(customer,
bankAccountNumber);

        //Act
        bankAccount.WithdrawMoney(200, "the reason..");
    }

    [TestMethod()]
    [ExpectedException(typeof(InvalidOperationException))]
    public void BankAccountCannotDepositMoneyInLockedAccount()
    {
        //Arrange

        Customer customer = CustomerFactory.CreateCustomer("John", "Smith",
IdentityGenerator.NewSequentialGuid(), new Address("city", "zipcode", "AddressLine1",
"AddressLine2"));
        customer.Id = IdentityGenerator.NewSequentialGuid();

        string bankAccountNumber = "AB001";
        BankAccount bankAccount = BankAccountFactory.CreateBankAccount(customer,
bankAccountNumber);
        bankAccount.Lock();

        //Act
        bankAccount.DepositMoney(200, "the reason..");
    }
    //MORE TESTS
    //...
    //...
}

```

Table 15.- Domain layer tests



Rule N°: 17.

Always implement Unit Testing for the Domain Layer.

○ **Recommendation**

- Domain layer tests must be run in isolation from any dependency, such as databases.
- Check that all the tests are repeatable, that is, two sequential executions of the same test return the same result, without requiring a prior step.
- Avoid excessive test cleanup and preparation code since it might affect readability.



References

Unit Test Patterns: <http://xunitpatterns.com>