Tackling Complex Business Logic

In his book, Eric Evans presents a set of patterns aimed at **tightly relating the code to the**

**underlying model of the business domain**: aggregate, value objects, repositories, and

others. These patterns closely follow where Martin Fowler left off in his book and resemble

an effective set of tools for implementing **the domain model pattern.**

*I add-the next statement is a bit confusing, I think he’s saying that in order to have a design aligned with the domain, you don’t have to use DDD patterns. Use what then??:*

The patterns that Evans introduced are often referred to as tactical domain-driven

design. To eliminate the confusion of thinking that implementing domain-driven

design necessarily entails the use of these patterns to implement business logic, the author prefers to stick with Fowler’s original terminology. The pattern is “domain model,” and

the aggregates and value objects are its building blocks.

# Domain Model

The domain model pattern is intended to cope with cases of complex business logic.

**Here, instead of CRUD interfaces, we deal with:**

* complicated state transitions
* business rules, and invariants: rules that have to be protected at all times.

Let’s assume we are implementing a help desk system. Consider the following excerpt

from the requirements that describes the logic controlling the lifecycles of support

tickets:

• Customers open support tickets describing issues they are facing.

• Both the customer and the support agent append messages, and all the correspondence is tracked by the support ticket.

• Each ticket has a priority: low, medium, high, or urgent.

• An agent should offer a solution within a set time limit (SLA) that is based on the

ticket’s priority.

• If the agent doesn’t reply within the SLA, the customer can escalate the ticket to

the agent’s manager.

• Escalation reduces the agent’s response time limit by 33%.

• If the agent didn’t open an escalated ticket within 50% of the response time limit,

it is automatically reassigned to a different agent.

• Tickets are automatically closed if the customer doesn’t reply to the agent’s questions within seven days.

• Escalated tickets cannot be closed automatically or by the agent, only by the customer or the agent’s manager.

• A customer can reopen a closed ticket only if it was closed in the past seven days.

These requirements form an entangled net of dependencies among the different rules, all affecting the support ticket’s lifecycle management logic. This is not a CRUD data entry screen, as we discussed in the previous chapter.

Attempting to implement this logic using active record objects will make it easy to:

* **duplicate the logic**
* **corrupt the system’s state by mis-implementing some of the business rules.**

## Implementation

**A domain model is an object model of the domain that incorporates both behavior and data**.

DDD’s tactical patterns—**aggregates**, **value objects**, **domain events**, and **domain services**—are the building blocks of such an object model. All of these patterns share a common theme: they put the business logic first. Let’s see **how the domain model addresses different design concerns:**

Attention: the patterns and concepts discussed here will use an OO language but they are applicable to the Functional paradigm too.

### Complexity

The domain’s business logic is already inherently complex, so the objects used for modeling it should not introduce any additional accidental complexities.

The model should be devoid of any infrastructural or technological concerns, such as implementing calls to databases or other external components of the system.

This restriction requires the model’s objects to be ***plain old objects***, objects implementing

business logic without relying on or directly incorporating any infrastructural components or frameworks

### Ubiquitous Language

The emphasis on business logic instead of technical concerns makes it easier for the

domain model’s objects to follow the terminology of the bounded context’s ubiquitous

language. In other words, **this pattern allows the code to “speak” the ubiquitous**

**language and to follow the domain experts’ mental models.**

# Building Blocks

Let’s look at the central domain model building blocks, or **tactical patterns, offered by DDD:** value objects, aggregates, and domain services.

## Value Objects

**A value object is an object that can be identified by the composition of its values**. For

example, consider a color object:

**class Color**{

**int** \_red;

**int** \_green;

**int** \_blue;

}

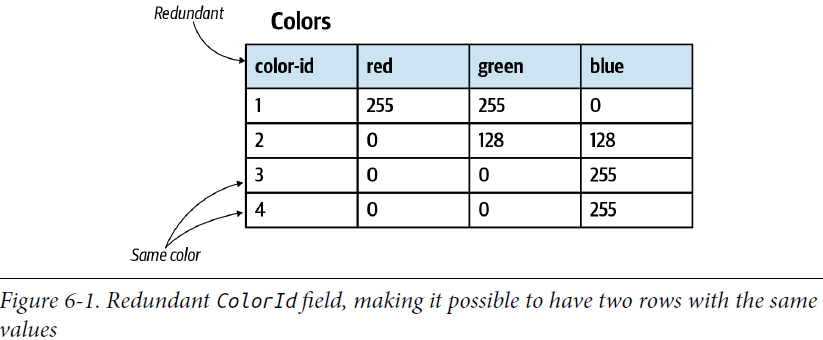
* The composition of the values of the three fields red, green, and blue defines a color.
* Changing the value of one of the fields will result in a new color.
* No two colors can have the same values.
* Also, two instances of the same color must have the same values.
* Therefore, no explicit identification field is needed to identify colors.

The ColorId field shown in Figure below is not only redundant, but actually creates an

opening for bugs. **You could create two rows with the same values of red, green, and**

**blue, but comparing the values of ColorId would not reflect that this is the same**

**color.**



### Ubiquitous Language

Relying exclusively on the language’s standard library’s primitive data types—such as strings, integers, or dictionaries—to represent concepts of the business domain is known as **the primitive obsession code smell**. For example, consider the following class:

**class Person**{

**private int** \_id;

**private string** \_firstName;

**private string** \_lastName;

**private string** \_landlinePhone;

**private string** \_mobilePhone;

**private string** \_email;

**private int** \_heightMetric;

**private string** \_countryCode;

**public** Person(...) {...}

}

**static void** Main(**string**[] args){

**var** dave = **new** Person(

id: 30217,

firstName: "Dave",

lastName: "Ancelovici",

landlinePhone: "023745001",

mobilePhone: "0873712503",

email: "dave@learning-ddd.com",

heightMetric: 180,

countryCode: "BG");

}

In the preceding implementation of the Person class, most of the values are of type String and they are assigned based on convention. For example, the input to the landlinePhone should be a valid landline phone number, and the countryCode should be a valid, two-letter, uppercased country code. Of course, the system cannot trust the user to always supply correct values, and as a result, **the class has to validate all input fields.**

This approach presents multiple design risks:

* First, **the validation logic tends to be duplicated**.
* Second, **it’s hard to enforce calling the validation logic before the values are used.** It will become even more challenging in the future, when the codebase will be evolved by other engineers.

Compare the following alternative design of the same object, this time leveraging

value objects:

**class Person** {

**private** PersonId \_id;

**private** Name \_name;

**private** PhoneNumber \_landline;

**private** PhoneNumber \_mobile;

**private** EmailAddress \_email;

**private** Height \_height;

**private** CountryCode \_country;

**public** Person(...) { ... }

}

**static void** Main(**string**[] args)

{

**var** dave = **new** Person(

id: **new** PersonId(30217),

name: **new** Name("Dave", "Ancelovici"),

landline: PhoneNumber.Parse("023745001"),

mobile: PhoneNumber.Parse("0873712503"),

email: Email.Parse("dave@learning-ddd.com"),

height: Height.FromMetric(180),

country: CountryCode.Parse("BG"));

}

* First, notice the increased clarity. Take, for example, the country variable. **There is no need to elaborately call it “countryCode” to communicate the intent of it holding a country code and not, for example, a full country name.** The value object **makes the intent clear, even with shorter variable names.**
* Second, there is no need to validate the values before the assignment, as the validation logic resides in the value objects themselves.
* a value object’s behavior is not limited to mere validation. Value objects shine brightest when they centralize the business logic that manipulates the values:

**The cohesive logic is implemented in one place and is easy to test**.

Most importantly, value objects express the business domain’s concepts: they make the code speak the ubiquitous language. (It’s not quite clear for me at the moment)

Let’s see how representing the concepts of height, phone numbers, and colors as value objects makes the resultant type system rich and intuitive to use.

Compared to an integer-based value, the Height value object both makes the intent

clear and decouples the measurement from a specific measurement unit. For example,

**the Height value object can be initialized using both metric and imperial units**,

making it easy to convert from one unit to another, generating string representation,

and comparing values of different units:

**var** heightMetric = Height.Metric(180);

**var** heightImperial = Height.Imperial(5, 3);

**var** string1 = heightMetric.ToString(); *// "180cm"*

**var** string2 = heightImperial.ToString(); *// "5 feet 3 inches"*

**var** string3 = heightMetric.ToImperial().ToString(); *// "5 feet 11 inches"*

**var** firstIsHigher = heightMetric > heightImperial; *// true*

The PhoneNumber value object can encapsulate the logic of parsing a string value, validating

it, and extracting different attributes of the phone number; for example, the

country it belongs to and the phone number’s type—landline or mobile:

**var** phone = PhoneNumber.Parse("+359877123503");

**var** country = phone.Country; *// "BG"*

**var** phoneType = phone.PhoneType; *// "MOBILE"*

**var** isValid = PhoneNumber.IsValid("+972120266680"); *// false*

The following example demonstrates the power of a value object when it encapsulates

all of the business logic that manipulates the data and produces new instances of the

value object:

**var** red = Color.FromRGB(255, 0, 0);

**var** green = Color.Green;

**var** yellow = red.MixWith(green);

**var** yellowString = yellow.ToString(); *// "#FFFF00"*

As you can see in the preceding examples, value objects **eliminate the need for conventions**— for example, the need to keep in mind that this string is an email and the other string is a phone number—and instead **makes using the object model less error prone and more intuitive.**

### Implementation

**Since a change to any of the fields of a value object results in a different value, value objects are implemented as immutable objects.(But Why?)**

**A change to one of the value object’s fields conceptually creates a different value—a different instance of a value object.**

Therefore, when an executed action results in a new value, as in the following case, which uses the MixWith method, it doesn’t modify the original instance but instantiates and returns a new one:

**public class Color**

{

**public readonly byte** Red;

**public readonly byte** Green;

**public readonly byte** Blue;

**public** Color(**byte** r, **byte** g, **byte** b){**this**.Red = r; **this**.Green = g; **this**.Blue = b;}

**public** Color MixWith(Color other)

{

**return new** Color(

r: (**byte**) Math.Min(**this**.Red + other.Red, 255),

g: (**byte**) Math.Min(**this**.Green + other.Green, 255),

b: (**byte**) Math.Min(**this**.Blue + other.Blue, 255));}

}

**Since the equality of value objects is based on their values rather than on an id field or reference, it’s important to override and properly implement the equality checks.**

For example, in C#:

**public class Color**{

**public override bool** Equals(**object** obj){

**var** other = obj **as** Color;

**return** other != **null** &&

**this**.Red == other.Red &&

**this**.Green == other.Green &&

**this**.Blue == other.Blue;

}

**public static bool operator** == (Color lhs, Color rhs)

{

**if** (Object.ReferenceEquals(lhs, **null**)) {

**return** Object.ReferenceEquals(rhs, **null**);

}

**return** lhs.Equals(rhs);

}

**public static bool operator** != (Color lhs, Color rhs)

{

**return** !(lhs == rhs);

}

**public override int** GetHashCode()

{

**return** ToString().GetHashCode();

}

}

**This needs more clarification:**

Although using a core library’s Strings to represent domain-specific values contradicts

the notion of value objects, in .NET, Java, and other languages **the string type is**

**implemented exactly as a value object**. Strings are immutable, as all operations result

in a new instance. Moreover, the string type encapsulates a rich behavior that creates

new instances by manipulating the values of one or more strings: trim, concatenate

multiple strings, replace characters, substring, and other methods.

### When to Use Value Objects

The simple answer is**, whenever you can**.

**GET BACK TO THESE BENIFITS:**

* Not only do **value objects make the code more expressive**
* and **encapsulate business logic that tends to spread apart**
* but the **pattern makes the code safer.** (*I think he refers to the centralized validation logic, or preventing some piece of code changing the values where a reference to that value object is being used somewhere else in the code*)
* **Since value objects are immutable, the value objects’ behavior is free of side effects and is thread safe.**

From a business domain perspective, a useful rule of thumb is to **use value objects for**

**the domain’s elements that describe properties of other objects**. **This namely applies to properties of entities**, which are discussed in the next section.

The examples you saw earlier used value objects to describe a person, including their ID, name, phone numbers, email, and so on. Other examples of using value objects include various statuses, passwords, and more business domain–**specific concepts that can be identified by their values** **and thus do not require an explicit identification field.**

**An especially important opportunity to introduce a value object is when modeling money and other monetary values. Relying on primitive types to represent money not only limits your ability to encapsulate all money-related business logic in one place, but also often leads to dangerous bugs, such as rounding errors and other precision-related issues.**

## Entities

An *entity* is the opposite of a value object. **It requires an explicit identification field to**

**distinguish between the different instances of the entity**. A trivial example of an entity

is a person. Consider the following class:

**class Person**{

**public** Name Name { **get**; **set**; }

**public** Person(Name name){

**this**.Name = name;

}}

The class contains only one field: name (a value object). This design, however, is suboptimal

because **different people can be namesakes and can have exactly the same names. That, of course, doesn’t make them the same person. Hence, an identification field is needed to properly identify people:**

**class Person**{

**public readonly** PersonId Id;

**public** Name Name { **get**; **set**; }

**public** Person(PersonId id, Name name){

**this**.Id = id;

**this**.Name = name;}

}

In the preceding code, we introduced the identification field Id of type PersonId.

**PersonId is a value object**, and it can use any underlying data types that fit the business

Domain’s needs. For example, the Id can be a GUID, a number, a string, or a domain-specific value such as a Social Security number.

**The central requirement for the identification field is that it should be unique for each instance of the entity:** for each person, in our case.

The identifier lets two instances represent two different People even though they have the exact same name, etc.

Furthermore,except for very rare exceptions,**(Like what?)** the value of an entity’s identification field shouldremain immutable throughout the entity’s lifecycle. This brings us to the second conceptual difference between value objects and entities:

**Contrary to value objects, entities are not immutable and are expected to change.**

Another difference between entities and value objects is that **value objects describe an**

**Entity’s properties.** Earlier in the chapter, you saw an example of the entity Person and

it had two value objects describing each instance: PersonId and Name.

Entities are an essential building block of any business domain. That said, you may

have noticed that earlier in the chapter **he didn’t include ”entity” in the list of the domain model’s building blocks**. That’s not a mistake. **The reason “entity” was omitted**

**is because we don’t implement entities independently, but only in the context of the aggregate pattern.**

*I add: the differences between Entities and Value Objects:*

* *Unlike Value Objects that are identified by the composition of their filed values, Entities need explicit Identifiers and two Entities with the same properties (except for ID) represent two different instances.*
* *Unlike Value Objects, Entities are expected to change and are not immutable*
* *Value Objects are used to describe Entities*

## Aggregates

**An aggregate is an entity**: it requires an explicit identification field and its state is

expected to change during an instance’s lifecycle.

However, it is much more than just an entity. **The goal of the pattern is to protect the consistency of its data.** Since an aggregate’s data is mutable, there are implications and challenges that the pattern has to address to keep its state consistent at all times.

### Consistency Enforcement

**Since an aggregate’s state can be mutated, it creates an opening for multiple ways in which its data can become corrupted.**

To enforce consistency of the data, **the aggregate pattern draws a clear boundary between the aggregate and its outer scope:** **the aggregate is a consistency enforcement boundary.** **The aggregate’s logic has to validate all incoming modifications and ensure that the changes do not contradict its business rules.**

From an implementation perspective**, the consistency is enforced by allowing only**

**the aggregate’s business logic to modify its state. All processes or objects external to**

**the aggregate are only allowed to read the aggregate’s state. Its state can only be mutated by executing corresponding methods of the aggregate’s public interface.**

The state-modifying methods exposed as an aggregate’s public interface are often

referred to as commands, as in “a command to do something.”

A command can be implemented in two ways. First, it can be implemented as a plain public method of the aggregate object:

**public class Ticket**{

**public void** AddMessage(UserId **from**, **string** body){

**var** message = **new** Message(**from**, body);

\_messages.Append(message);

}}

Alternatively, a command can be represented as a [parameter object](https://oreil.ly/4hNtn) that encapsulates

all the input required for executing the command:

**public class Ticket**{

**public void** Execute(AddMessage cmd){

**var** message = **new** Message(cmd.**from**, cmd.body);

\_messages.Append(message);

}}

How commands are expressed in an aggregate’s code is a matter of preference. I prefer

the more explicit way of defining command structures and passing them polymorphically

to the relevant Execute method.(What do you mean by polymorphic? I think he means overloading and .NET terms are different)

**An aggregate’s public interface is responsible for validating the input and enforcing all of the relevant business rules and invariants**. This strict boundary also ensures that **all business logic related to the aggregate is implemented in one place: the aggregate itself.**

86……