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;

}

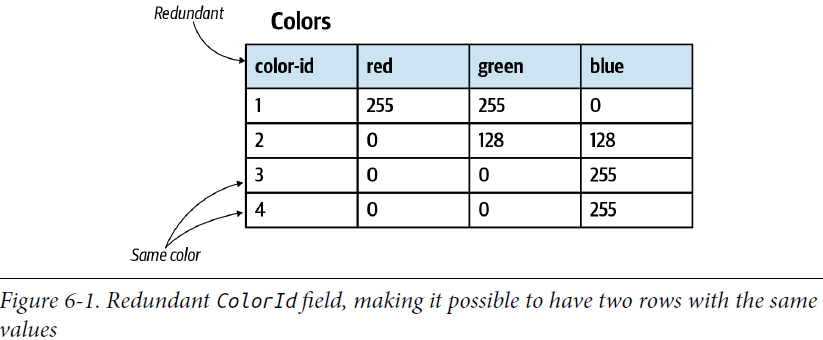
* The composition of the values of the three fields red, green, and blue defines a color.
* Changing the value of one of the fields will result in a new color.
* No two colors can have the same values.
* Also, two instances of the same color must have the same values.
* Therefore, no explicit identification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