Using a DDD Approach for Validating Business Rules

Leia em PortuguÃªs

This item in japanese

Key Takeaways

- Business rules express part of the knowledge a system has about a specific domain.
- Structuring business rules appropriately reduces the possibility of error and makes the maintenance easier.
- Business rules have different responsibilities, such as providing validation or generating outputs.
- Deciding how to process events has an impact on the general design and the consequences need to be understood before making any decision.
- Using a command approach should be structured around tasks a user executes and provide a clear output.

When you think about business rules, especially if you are a technical person, you probably imagine lines of code or a business rules engine (if you use one). However, rules are more related to knowledge management than coding. For instance, a person working in the construction sector knows there are many rules about the minimum size of a bedroom or what are required characteristics for a space to be considered a bedroom instead of a den. Knowing the rules and how to apply them make this person an expert in this domain. Our goal is to create software applications that emulate the behavior of domain experts.

Just by reading this first sentence, you probably can generate a quick list of questions like, what is a resource? What does it mean to book a resource? Does the system allow a resource to be booked more than once? Questions like these should be answered by the domain expert and will be the business rules that control the behavior of the system.

This also will guide the definition of entities, value objects, commands, and other elements of the system. How to define each element is beyond the scope of this article.

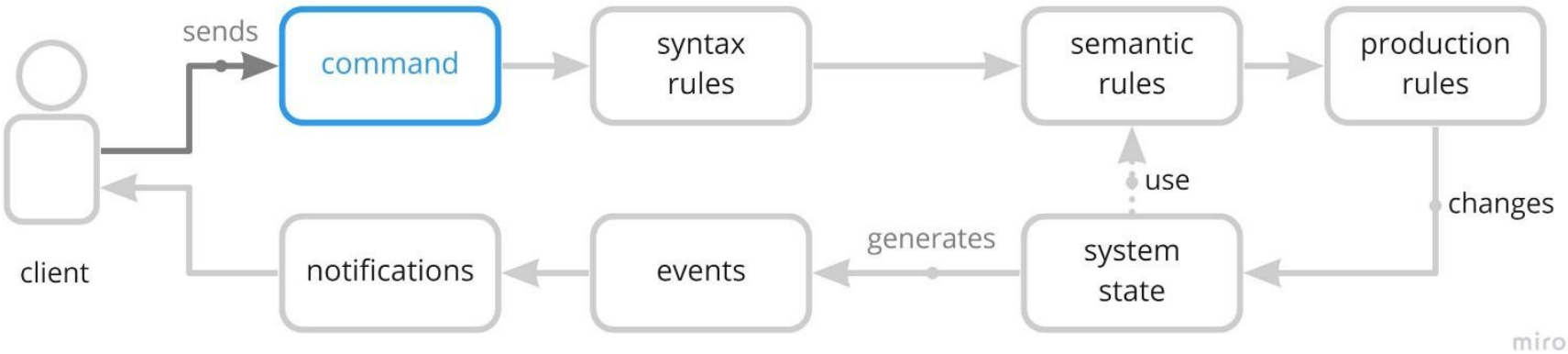
There are books dedicated to this topic like Eric Evans' *Domain-Driven Design*, and if you are implementing a .NET I would recommend Hands-On Domain-Driven Design with .NET Core.

This article will cover only the part of how to structure business rules that:

1. **Are consistent**. Organizing rules in a way that any developer can understand is paramount for any future change. It would be best if you considered that systems always change and that applications should be prepared for such changes.
2. **Are easy to test**. Making testable rules is the only way to guarantee that changes are not going to break existing functionalities.
3. **Maintain data consistency**. Keeping data consistent is one of the primary responsibilities of the business rules. In my personal experience, many data quality issues are related to poor handling of the system rules.
4. **Are easy to diagnose**. We should aim to avoid any unexpected error; however, exceptions are always going to happen. Such errors are even worse when the only diagnostic information you have says **object reference null**.

can perform any action, it must validate that the provided information is valid. The next sections describe how to handle these validations.

Detailed model



Base model for handling validations using a domain-based approach - here showing a command sent to the system

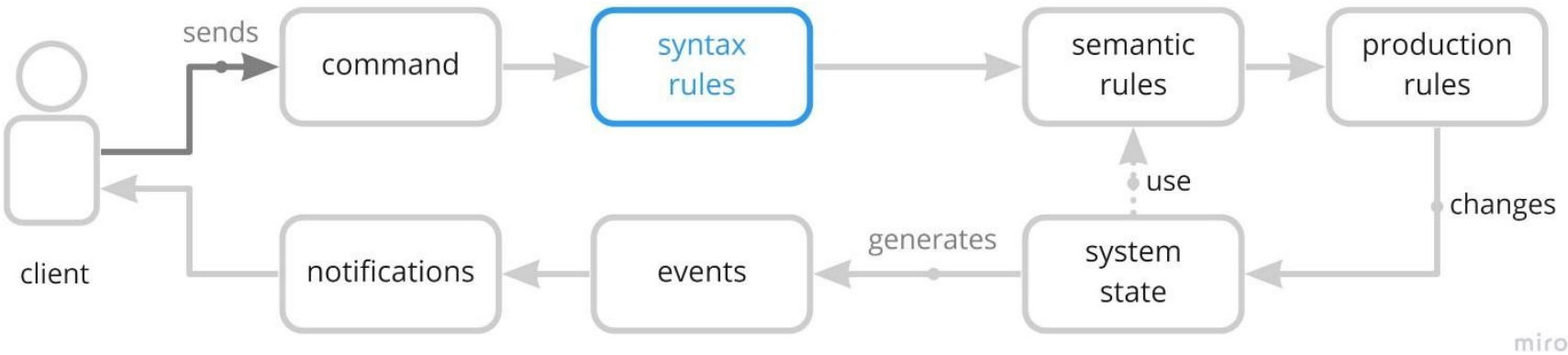
This model provides an example of the implementation of handling business rules within a system. This approach is based on domain-driven design (DDD), but it can be applied without using it as well. The sequence of execution is as follows:

Command sent to the system

1. A client sends a command

middle point between software design and business design. It may sound trivial, but when it's specified, it helps us to understand a system design more efficiently. The idea connects with the HCI (human-computer interaction) concept of designing systems with a task in mind; the command helps designers to think about the specific task that the system needs to support.

The command may have additional parameters, such as date, resource name, and description of the usage. Command and Query Responsibility Segregation (CQRS) is a pattern that uses commands for modeling a system interaction. Although in the model image, the client is a user, it could be another system as well. It also allows the design of the system to have different behaviors for actions that change the state of the system from those that only return data.

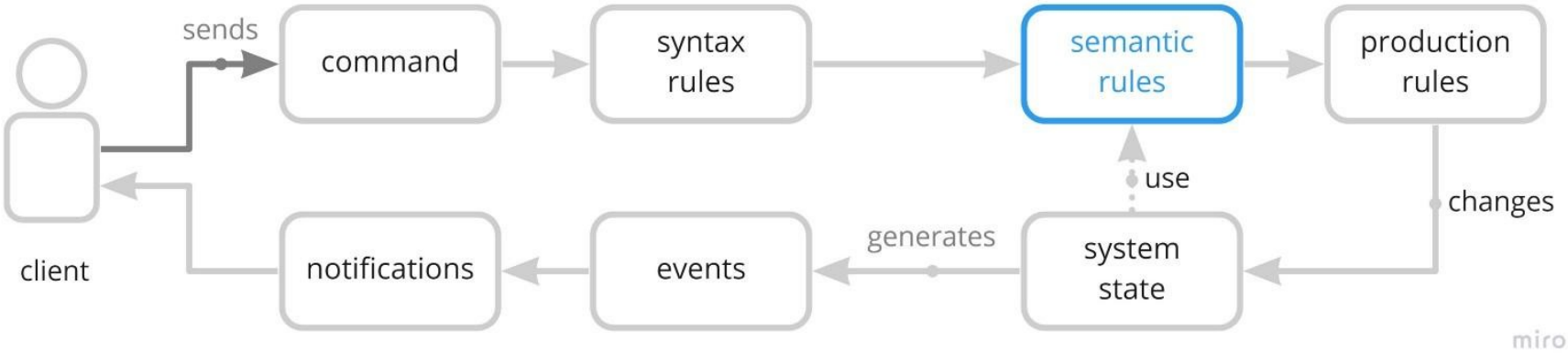


Syntax validation of a command

2. Syntax rules validation

command makes sense in the context of the domain.

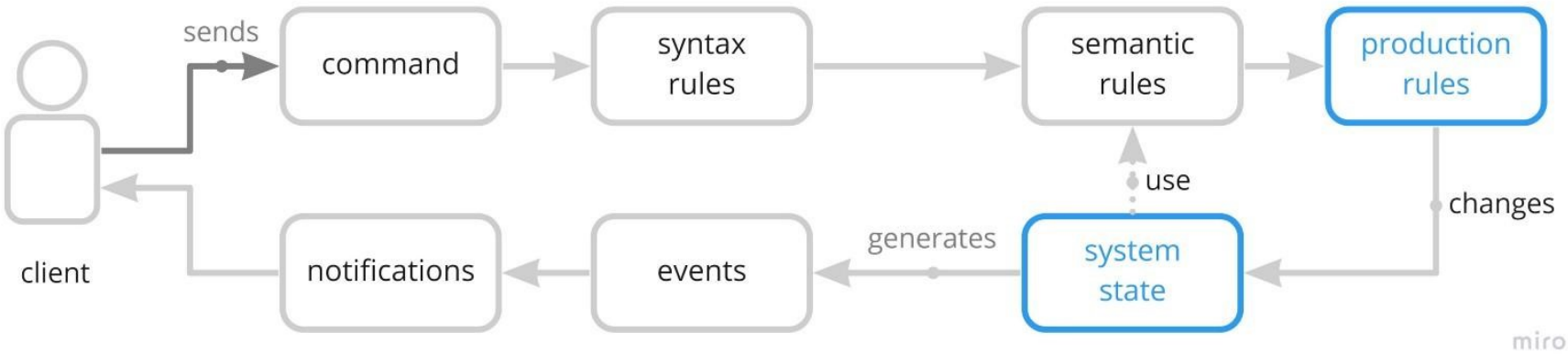
This validation guarantees that the provided data is coherent for the system. One example of this validation is for the start date for making a reservation - its value shouldn't be in the past, and cannot be null. There are different ways to return an error during this validation stage; Vladimir Khorikov provides some examples of how to perform these validations. This validation can happen in different layers of the system as well. For example, it is better to validate in the user interface that past dates can not be provided instead of sending these values for validation in the back-end service. It does not mean you shouldn't have it in the domain as well; it prevents any misuse by a different client.



Semantic rules validation using the system state

3. Semantic rules validation

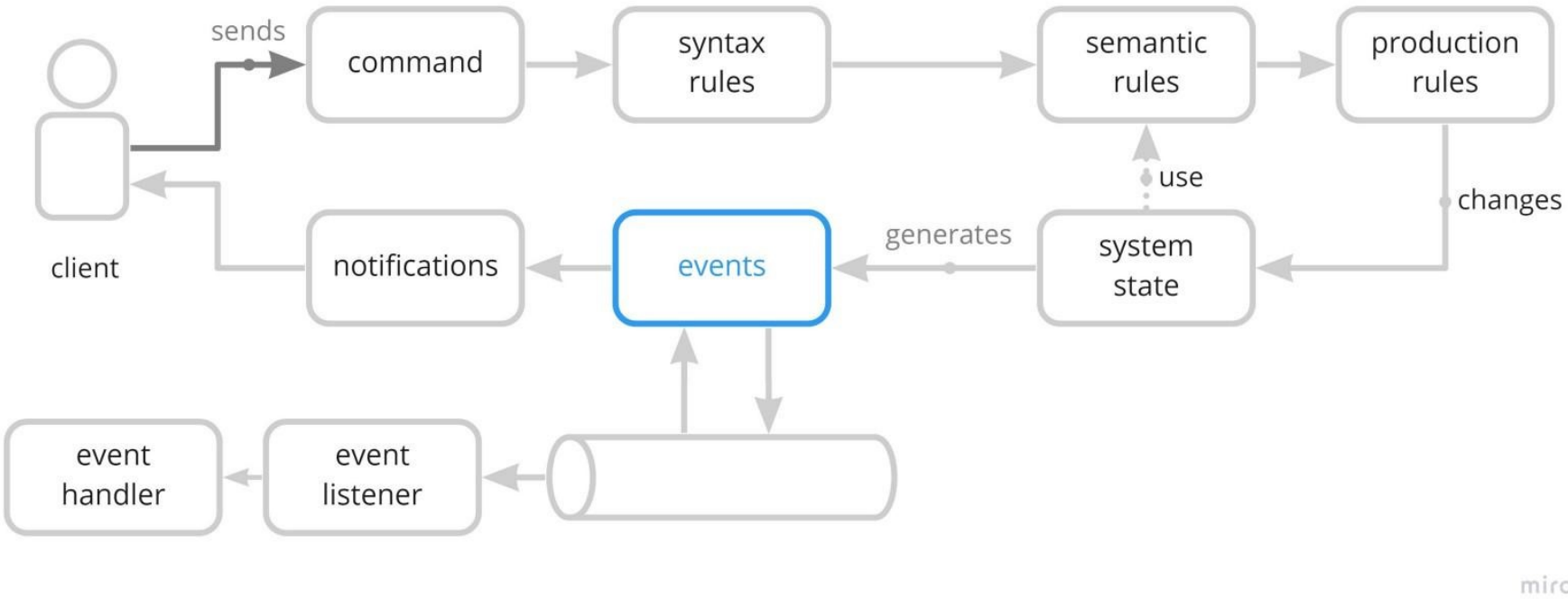
the MakeReservation command, one of such validations is to check that another customer has not reserved the requested resource. This validation can be in a client-side application as well; nonetheless, there are multiple reasons why the domain can still receive a request to process a reservation for the same resource. The responsibility of the system is to ensure data consistency.



Processing production rules that change the system state

4. Production rules

achieve the desired state. They deal with the task a client is trying to accomplish. Using the MakeReservation command as a reference, they make the necessary changes to register the requested resource as reserved. As a consequence of these modifications, entities and domain services can generate events that let the client and other systems know that a change has occurred. In the example we have been using, it would be ResourceReserved.



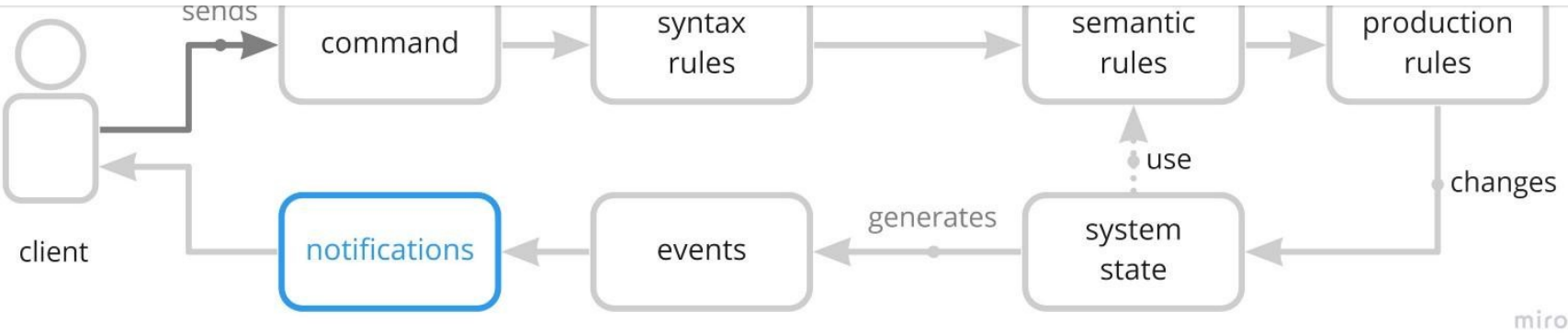
Events generation and processing

5. Events

for processing events. Each one offers its pros and cons.

The first one is by using transactional consistency. In this model, the system keeps the coherence of the modified entities by using the same transaction. Even when there is more than one aggregate involved, the transaction keeps track of the changes and persists them at the end of the process. This pattern is known as a unit of work. It provides certain advantages like simplicity and efficiency in terms of code production. However, sometimes it is impractical to use it. One of the reasons is that it keeps the modified entities in memory, which, depending on the load, will use too many memory and processing resources. Additionally, if you are communicating outside of the domain, you will need to use distributed transactions, making the application even more complicated.

The second option is to use eventual consistency. This enhances the system capacity because you are not dealing with a big entity graph in memory while executing your command. This decision, however, has other consequences. It implies that the reservation system completes its changes, and if other subsystems report a failure, the reservation system must implement a compensation transaction that will revert the previous operation. It also encompasses notifying the user about potential errors. The decision to use one or the other should weigh the pros, cons, cost of development, and the required infrastructure. For example, it could require a message broker for sending and receiving messages from and to other domains.



Sending notifications to clients

6. Notifications

Once the command is completed, the system has to inform the client about the result of the execution. If the client is another system, then this part can be omitted. Yet, if it is an application, we probably need to notify the user about the success or failure of a command execution, especially when there is asynchronous processing involved. Continuing the MakeReservation command, one of the requirements is that specific resources need the owner's approval before they can be assigned. Once the resource usage is approved, the system must send a confirmation to the user. The notification in this model starts when the event ReservationApproved is generated. This event is the result of another command which should follow the same logic we described before, but a different user executes it.

The idea of including this in the design is that the user task is not completed unless the approval is granted. Also, if the permission is denied, the domain will probably need to revert part of the changes. When designing a system, we have to consider all interactions until the user's task is completed.

Business rules are the way that applications express the knowledge they have about a domain. Proper structuring of business rules helps developers easily understand the code, and should be divided into different categories according to their responsibility. Also, structuring the code to represent user tasks reduces the number of concepts the development team has to understand. Because business rules depend on the system design, a solutions architect must consider how the design will support them.

About the Author

Fabian Lopez is a software engineer doing a master's in computer sciences at Georgia Tech. Currently he works as a solutions architect at the Inter-American Development Bank, designing process management applications in the baking industry. Lopez likes to use and create software patterns because it provides a reference point for solving many development issues. He is regularly involved in user research to identify user patterns and develop solutions that work for customers. Lopez has noticed in the past that many good technical solutions failed because users were lost while using applications, which generates a poor perception of the solution quality.

Discuss