ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/370471813

On business logic layer design and architecture.

Article *in* System technologies · February 2020

CITATIONS
0

READS
88

1 author:

Oleksandr Lytvynov
Oles Honchar Dnipro National University
20 PUBLICATIONS **7** CITATIONS

SEE PROFILE

All content following this page was uploaded by Oleksandr Lytvynov on 03 May 2023.

The user has requested enhancement of the downloaded file.

УДК 614.2+574/578+004.38

Litvinov A.A.

BUSINESS LOGIC LAYER DESIGN AND ARCHITECTURE

Annotation. The work is devoted to business logic layer construction using Clean Architecture as a foundation. Additional layer of Activities is introduced to make Clean Architecture more effective for business-process oriented domain modeling, making software more flexible and robust, testable and maintainable.

Key words: domain-driven design, business logic layer, clean architecture.

Importance and relevance of the research topic. Decade-to-decade and even year-to-year the complexity of software keeps growing. Business requires more than just an application or even a system meets the needs. Today's business requires more flexible and agile, maintainable and testable, robust and scalable infrastructure able to be an asset. How to design, realize and maintain such systems? What principles, architecture and patterns should be used by the developers to tackle that overwhelming complexity? These questions are stated before developers, managers, scientists.

Undoubtedly, multi-layered architecture becomes a standard for modern information systems design. According to the separation of concerns principle each layer has its own responsibility: user application provides user an ability to interact with the system, web-service provides an ability of remote access to the system, data access layer enables the persistence of objects of the system. And where is the system itself? In short, the system is business logic layer. That is why business logic layer can be regarded as the heart of the system, the most important unit, which can be thought as a main processor interacting with the peripherals/infrastructure represented by other layers. And consequently, there is a problem of how to build this unit making it flexible, adoptable, maintainable, testable etc.

Analysis of recent publications. We can say that Domain-driven design (DDD) [1], Hexagonal, Onion [2] and Clean architectures [3] are responses to that question.

DDD (published in 2004 by Eric Evans) provides a number of principles and patterns allowing developers to tackle the complexity connected to busi-

© Litvinov A.A., 2020

ISSN 1562-9945

ness layer design. It declares that the focus point is the domain which should be examined thoroughly and then the result of such examining is a layer separated into two main horizontal parts: application layer (thin layer of services responsible for orchestrating user actions using domain objects) and domain layer responsible for representing concepts of the business, information about the business situation, and business rules. In accordance with Evans domain layer is the heart of the system. It contains domain events and their handlers, value objects, entities, aggregates etc. In addition, Evans declares that the objects should be behaviorally rich, and context bound.

Bounded Context is the central pattern of DDD. It is focused on strategic decomposition dealing with large models and teams. According to this pattern large models should be divided into different Bounded Contexts with explicit interrelationships. This keeps the knowledge inside the boundary consistent whilst ignoring the noise from the outside world. It helps to model the aspects of the problem (within the context) without having to be concerned with other parts of the business. Secondly, the terminology within the Bounded Context can have clear definition that accurately describe the problem whilst different departments across a company usually have slightly different ideas and definitions of similar terms. For example, for a standard sales company its domain could be divided into Sales, Support and Delivery contexts. Of course, the various Bounded Contexts of an application will need to communicate or share data between each other. That problem resolved using Context Map which defines how the Contexts should communicate amongst each other and how data should be shared (i.e. when a business process functions on different Contexts, Context Map is used to coordinate their interaction). There are different patterns used to realize such interactions: shared kernel, upstream-downstream, conformist, anticorruption layer etc. Evans has also introduced several conceptions widely used by software programmers such as: ubiquitous language; identity map, value object, repository, aggregate, event sourcing, specification and other patterns. The scopes of DDD was defined by Eric Evans as follows. DDD is the best applicable when there is a lot of business domain complexity and DDD is not suitable for problems with substantial technical complexity without business domain complexity. Thus, the main disadvantage of this approach is a complexity in domain anal-

ysis required a lot of time sent by an experienced architects and business analysts.

Hexagonal (ports and adapters) architecture (documented in 2005 by Dr. Alistair Cockburn) is a three-layered architecture: application, business logic(domain) and infrastructure. Originally, there is only one layer responsible for business logic, but in reality, it is divided into two sub-layers: application services and domain. But the architecture doesn’t matter on what is placed within the domain unit, but mostly concentrated on how that layer interacts the environment. The reason is pointed out by the slogan “allow an application to equally be driven by users, programs and automated tests, and to be developed and tested in isolation from its eventual run-time devices and databases”. The solution provided by Dr. Cockburn based on using the ports and adapters mechanism. There are two kinds of ports: ports and driven ports. Driver ports define use case boundary of the application. Actors interact with the driver ports not with the application itself. Practically, driver ports are the interfaces that the application offers to the outside world allowing actors interact with the application. Driven ports declare an interface for a functionality, needed by the domain layer for implementing the business logic. Driver ports define API (application programming interface) of the application and driven ports define SPI (service provider interface). Adapter is a software component that allows a technology to interact with a port of the hexagon. Thus, driver adapter uses a driver port interface converting a specific component request into an agnostic request to a driver port.

Onion architecture was inspired by hexagonal and created by Jeffrey Palermo in 2008 and provides more structural architecture than its predecessor with domain layer placed in the center and a number of layers separated it from infrastructure. Thus, domain layer could not be connected to infrastructure directly. The layers practically used are as follows: application services layer, domain services layer (i.e. repositories layer), domain layer.

Next one is clean architecture provided in 2012 by Robert C. Martin. Martin summarized the experience of the above approaches and provided an architecture based on ports and adapters with two layers of business logic: entities and use cases. Use cases is a layer of services responsible for business processes modeling and domain entities responsible for request, response,

rules and entities representation.

On the other hand, service-oriented architecture (SOA) [5] is based on the same principles as domain-driven approaches but used for enterprise integration systems development. Structurally, services are divided into three basic layers (instead of two basic layers used in domain-driven approaches): workflow services (equivalent of business process), task (equivalent to business task) and entity (equivalent to business object) services. And this model is intuitively closer to the models used by the business analysts to describe the business processes. But SOA principles are focused on how to glue the system from existed heterogenous components and services, and they could be applied directly for business logic layer development.

Task definition. Despite the differences among approaches they share the most important and most valuable common feature that is the principle aimed to build the software as a model of business which should represent the business as closer as it is possible. Does the model created according with the principles of DDD and its descendants satisfy modern business governed by business processes management approach? When we talk about business processes, we also talk about business functions and events, business objects and goals, outcomes and resources. Do all those abstractions can be represented using DDD effectively? It seems, does not.

Main part. To analyze the problems of provided approaches we should first understand what is the domain to which the infrastructure (i.e. information system) is applied. Since 1990 most of the leading business companies work using business process management model approach. It was provided by Dr. Davenport in earlier 1990s [4] and focused on the process, which is a structured, defined and measured set of activities designed to produce a specific output valuable for a customer or market. Business process management life cycle starts with design stage, in which the activities are analyzed and “as is” models are built. Despite the differences in approaches two basic forms remain the same: **building the catalogue of business functions and constructing business processes as compositions of business functions**. Business functions connected to different types of resources, and business processes are triggered by business events producing an output (BPMN 2.0 used to describe business process). Thus, according to R. Martin mapping

business model to software results in building use cases reflected user-system interaction scenarios. Of course, such mapping is focused primarily on information flows and objects. We can say that naturally business function is mapped to use case (i.e. class of application layer). For example, for a simple process shown in Fig.1 we will get the following sequence of use cases: Place Order, Identify Customer, Register Customer Account, Notify on Delivery, Pay for Delivered Goods. Process engages several departments.

But if we take a look at use cases, we will see that whilst the business functions are atomic from business point of view, use cases are not atomic from software development perspective. It is obvious, because the core of use case is a bunch of scenarios: basic one and alternatives [6]. And getting business functions mapped we don’t get atomic activities catalogue for software development, i.e. set of atomic actions involved in use case scenarios. And the lack of such things results in ineffective representations of use cases, betting on domain objects rather than atomic actions. We can say that use cases are composed of activities in such a way as business process composed of business functions. And in a way we can think about business functions as building blocks of the processes and similar activities shared among the processes, we can think about the activities involved in use cases. But all the provided approaches don’t have activities blocks separated in a layer, smearing them along use cases and domain layers.

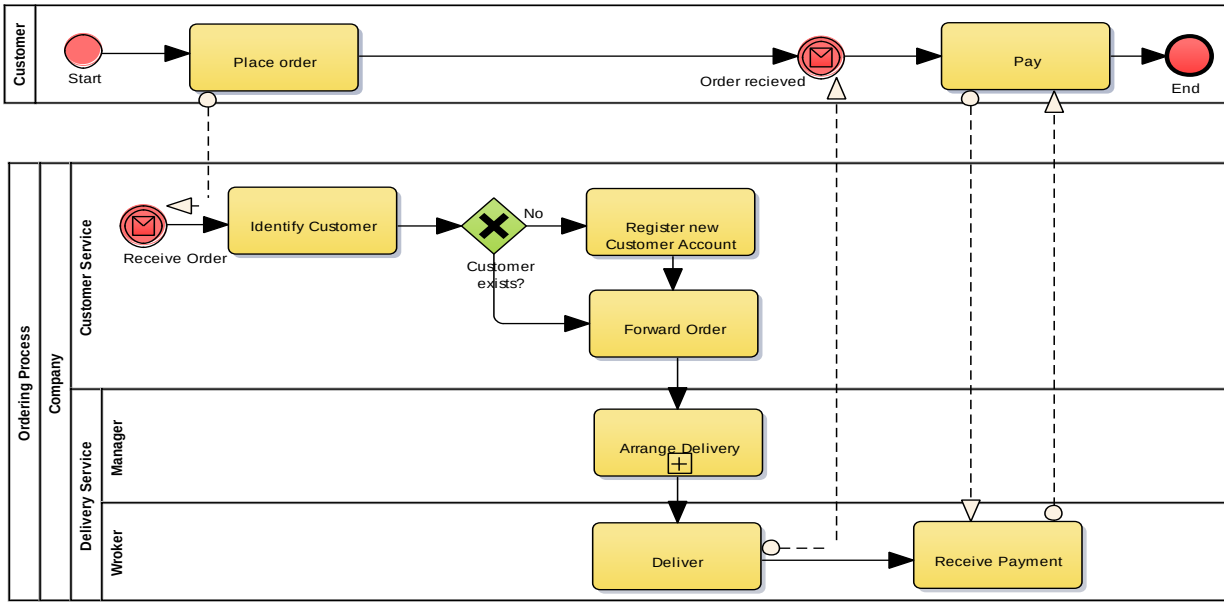

Figure 1. – Simple business process in BPMN 2.0 notation.

And that is the reason why we introduce an additional layer to two basic layers proposed by R. Martin in Clean Architecture, the **layer of Activities** (i.e. layer of Activities). The layer of Activities depends on infrastructure layer, because, as it was mentioned above, the functions naturally depend on resources provided by the infrastructure. Thus, use cases layer remains thin as it was suggested by Eric Evans, but is not realized in Clean Architecture (use cases depend on infrastructure such as repositories etc.). The model of software becomes more comprehensible, making the software more flexible and robust, testable and maintainable. The structure of business logic layer is shown in Fig. 2.

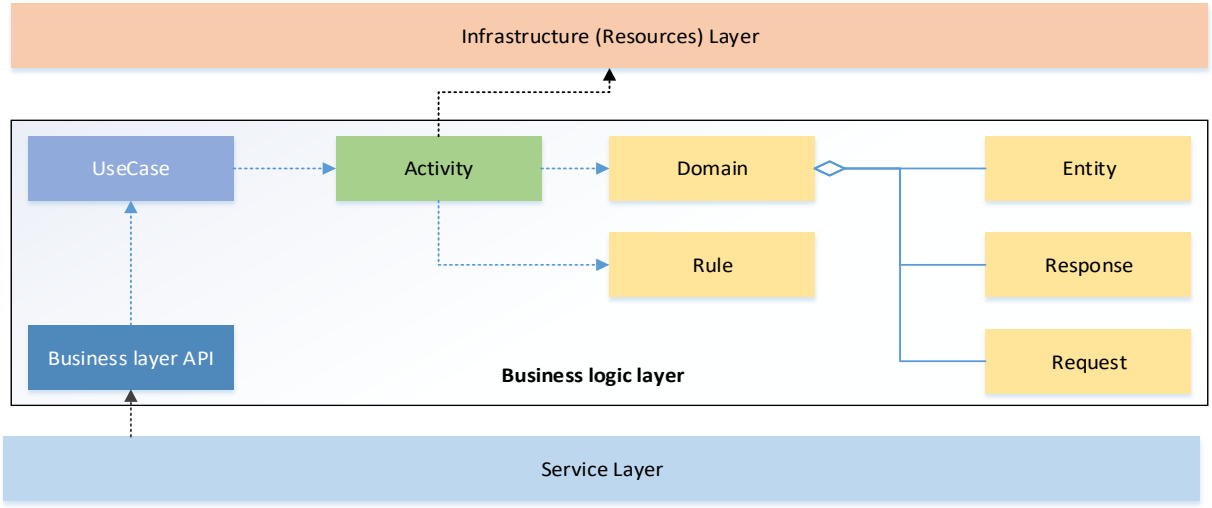

Figure 2. – The structure of business logic layer

Activities depend on domain objects and business rules. From that point of view, domain objects can be regarded as anemic domain objects. It is known subject of discussion and there are several opinions, including Fowler’s “the fundamental horror of this anti-pattern is that it's so contrary to the basic idea of object-oriented design; which is to combine data and process together” and his proposition to place the logic (validations, calculations, business rules) in domain objects (e.g. Book, Student). But if we try to apply an analogy of “processes-functions” to use cases, it seems that we will not find in business model any full-blooded business objects interacting with infrastructure.

When we try to make activity classes, we face a huge amount of similar logic shared among these activities and to exclude such complexity we need a kind of normalization. The way we propose is to make a set of basic generic activities which can be further used as a foundation for business-oriented activities creation, significantly simplifying their definition. Simply speaking, the most of business-oriented activities are inherited from the generic ones. Activity factory separates the layer from the layer of Use cases, allowing to create only business-oriented activities used by use case objects. We think it should be rather flexible to make the layer of use cases depended on activity factory interface then a hard-coded realization, allowing to use different realizations of activities (Fig. 3).

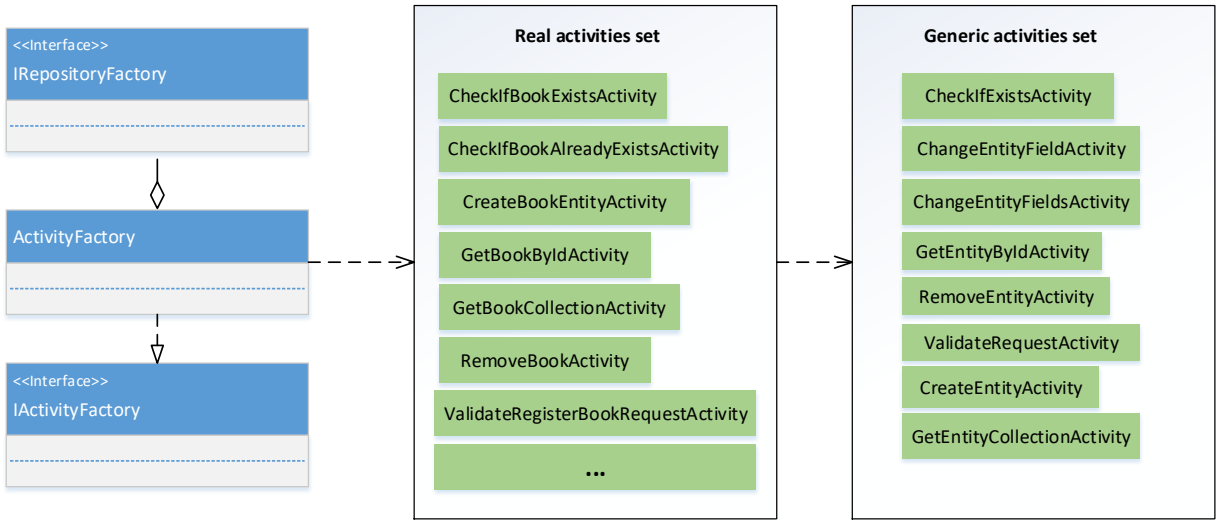

Figure 2. – Activity layer structure

Activities depend on resources such as repositories and external services. It is obvious that such dependencies should be resolved by dependency injection according to SOLID principles. A typical structure of activities is shown in Fig. 4. Use case factory creates realizations of use cases, injecting necessary activities provided by Activity factory. Obviously, use case factory depends on Activity factory realization. Typical use case structure is shown in Fig. 5.

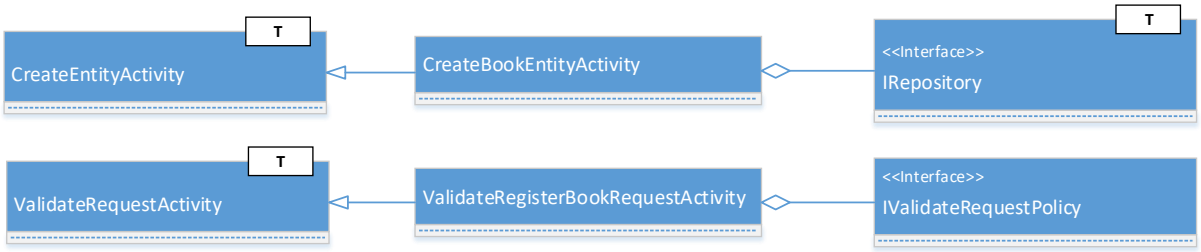

Figure 4. – Typical structure of activities

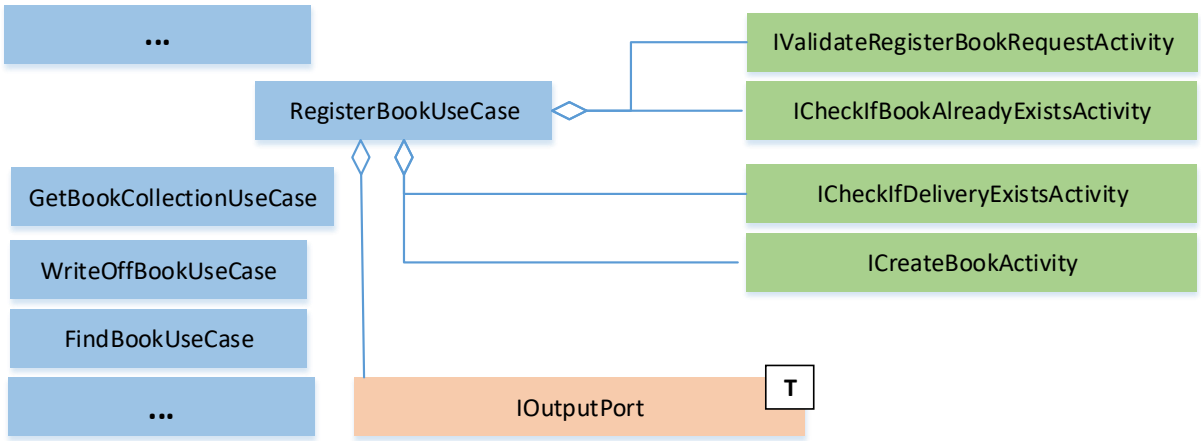

Figure 5. – Typical structure of use-case

An example of use case execution method is shown in Listing 1. It is notable that the request object (RegisterBookRequest) is walking through the bunch of activities to the point when a response object is constructed, using partial results obtained during activities execution, and then passed to the outputPort sink according to ports/adapters pattern suggested by Dr. Cockburn. Thus, use case plays a role of activities orchestrator (i.e. analog of workflow service in SOA) and it’s not directly depended on infrastructure components such as repositories.

Listing 1

```
@Override
public void execute(RegisterBookRequest registerBookRequest)
       throws Exception
{
this.validateRegisterBookRequestActivity.run(registerBookRequest);
this.checkIfBookAlreadyExistsActivity.run(registerBookRequest);
this.checkIfDeliveryExistsActivity
                     .run(registerBookRequest.getDeliveryId());
int id = this.createBookEntityActivity.run(registerBookRequest);
```

```
return outputPort.Post(new RegisterBookResponse(id));
}
```

Such computing description can be easily transformed into asynchronous variant using pipeline pattern. Construction of such pipeline will be based on wrapping the activities in pipeline stages. Simple variant of the pipeline is shown in Fig.6. Each stage is composed of queue and activity elements and able to interact with the next stage by passing request and response objects to its source.

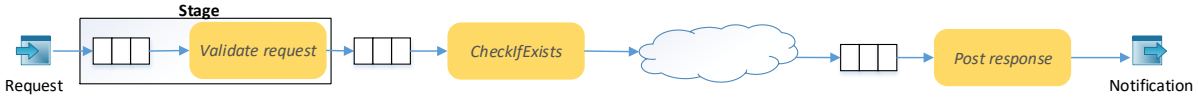

Figure 6. – Pipeline structure of use-case

Of course, use cases can have several conditional branches and loops that will be transformed into additional activities-stages with a number of outputs. We can think of such elements as resolvers responsible for reasoning which path to go. Resolvers are opposite to executors responsible only for performing actions. At first, it looks like providing complex solution to simple problems. But if we look at the solution with new eyes, we will see the solution becomes more comprehensible and structured, flexible and testable. The most important is that use cases become responsible only for activities orchestration, not depending on infrastructure, which make them very flexible.

Summary. The work is devoted to business logic layer construction using domain-driven design approach. Additional layer of Activities is provided to make Clean Architecture more effective for business-process oriented domain modeling. In the result the model of software becomes more comprehensible, allowing to make software more flexible and robust, testable and maintainable.

REFERENCES

1. Eric Evans. “Domain-Driven Design: Tackling Complexity in the Heart of Software”. 2003

2. J.Palermo. The Onion Architecture : part 1. https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/

3. Robert C. Martin. Clean Architecture. A Craftsman’s Guide to Software Structure and Design. 2018.

4. Davenport Thomas H. «Process Innovation: Reengineering Work through Information Technology» Published October 1st 1992 by Harvard Business Review Press. – 352 p.

5. Thomas Erl. Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall; 2016.

6. Cockburn A. Writing Effective Use Cases. / A. Cockburn – Addison-Wesley Professional. – 2000. – 304 p.

ISSN 1562-9945

View publication stats