

Domain-Driven Design Approaches in Cloud-Native Services Architecture

Jordan Jordanov ¹, Pavel Petrov ²

¹ University of Economics - Varna, Varna, Bulgaria (jordanov.jordan@ue-varna.bg)

² University of Economics - Varna, Varna, Bulgaria (petrov@ue-varna.bg)

Abstract – The number of cloud-based systems using Domain-Driven Design has been increasing in recent years. This paper gives a brief overview of domain-driven, cloud-based software development activities and how they fit into a well-known software development process. By giving a model based on theory, it emphasizes several techniques for expressing complex business logic. Sometimes conventional code base architectures are challenged by diversity, which transforms best practices into antipatterns. The significance of the system's availability, reliability, and resilience may prevent the organization from failure and support its growth. Domain-driven design demands that software code establish key principles such as "ubiquitous language" and "bounded contexts". In addition, a successful solution should contain command and query responsibility segregation, event sourcing patterns, and a comprehensive set of integration tests.

Keywords – domain-driven design, cloud computing, application programming interface, software development process, business logic complexity.

1. Introduction

Modern cloud-native services have an open application programming interface which facilitates online, desktop, and mobile clients to connect with one another.

DOI: 10.18421/TEMxx-xx

<https://doi.org/10.18421/TEMxx-xx>


Corresponding author: Pavel Petrov,
University of Economics - Varna, Varna, Bulgaria
Email: petrov@ue-varna.bg

Received: ----.

Revised: ----.

Accepted: ----.

Published: ----.

 © 2023. ----; published by UIKTEN. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 4.0 License.

The article is published with Open Access at <https://www.temjournal.com/>

The Domain-Driven Design (DDD) is an approach to software development that focuses on the application domain, its concepts, and their relationships as primary drivers for architecture design. Core principles of DDD include capturing relevant domain knowledge in domain models, which can include both structural and behavioral aspects, collaborative modeling between domain experts and software engineers. Experiment design is also encouraged by closely aligning model and implementation throughout the software development process, as well as continual model modification. (Sachweh, S., Zündorf, A., 2017). Domain-driven design provides patterns, activities, and examples of how to build a domain model, which is its main artifact (Hippchen, B. et al, 2017). Furthermore, including patterns and principles into software architecture concepts such as architecture views and their requirements assists software architects in designing cloud-native systems. (Giessler, P., Steinegger, R., 2017).

When creating a cloud solution, one of the first decisions to make is which service(s) to utilize in order to operate the applications. Table 1 shows the choices for which cloud services are best for which types of applications.

Table 1. Cloud services' suitability for various application types (Source: Rob Caron Sr. Product Marketing Manager, Barry Luijbregts, Microsoft Azure, 2022)

	Web service	Mobile service	Serverless	Virtual Machine	Microservices
Monolithic and N-Tier app	✓			✓	
Mobile app back end	✓	✓			✓
Distributed system			✓		✓
Business process workflow			✓	✓	

One of the simplest and most effective solutions of managing cloud-based app is the HTTP-based service for hosting web applications. Some examples

are Azure App Hosting Service, AWS Elastic Beanstalk, Google App Engine. They provide a set of services that host an application and hide the complexity of the operating system and infrastructure. They are highly available by default and will stay up and running for at least 99.95% of the time. They share powerful features like automatic scaling, zero-downtime deployments, and easy authentication and authorization (Blane, 2022). Some of them enable debugging the application while it is in production, using tools such as Snapshot Debugger (Hunter, 2022).

When developing a mobile application, a back end that the application can connect to is required. Typically, this is an API that the application can utilize to access and store data. Azure Mobile Apps and AWS Amplify provide such solutions with unique capabilities. For instance, there is an offline sync that enables the mobile app to continue working when there is no connection to the back end, and the sync changes once the connection is restored (Bahrami, 2015). Another feature is sending push notifications to the mobile apps, regardless of the platform they run on (iOS, Android, or Windows).

These programs are snippets of code written without concern for the underlying infrastructure or scalability. This deployment model is referred to as "Functions as a Service" (FaaS). in Roberts's study (2018). Even scaling is handled by these functions. They transparently spawn additional functions to handle heavy loads, and they disappear after the code has completed executing. Because of this, companies only pay for the code that is executed, not for a service that runs all the time, waiting to be triggered (Kumar & Agnihotri, 2021).

Existing applications could be lifted and relocated from virtual machines (VM) operating in a local datacenter to VMs running in the cloud, making this a simple approach to get started. There are many predefined VM images that are ready-to-use. However, running the application in a VM doesn't provide features like zero-downtime deployments or easy authentication. The operation team is also responsible for patching the operating system and making sure that antivirus software is up to date (Shah & Shah, 2018). Azure Virtual Machines, Amazon EC2 and Google Compute Engine are such solutions.

All the aforementioned types are created individually as monolithic large core application that contain all of the domain logic. It has components that communicate to one another directly within a single server process (Vettor, 2022). A monolithic application is a solitary, integrated unit, whereas microservices divide it into a number of smaller units.

Microservices are an organizational and architectural approach to developing software. According to this approach, software is composed of loosely connected services that are organized around business capabilities and that can be independently deployed and tested (Wolff, 2016). These services communicate with one another via well-defined APIs. Large, sophisticated applications may be delivered quickly, consistently, and reliably. Microservices are technology and language-agnostic, so it is quite possible for a single organization to utilize multiple runtime platforms. Modern cloud platforms have features like scalability, availability, and resilience that can be used to their fullest by microservices (Smith, 2022). Such cloud solutions are Azure Kubernetes Service, Amazon EC2 & EKS, Google Kubernetes Engine, Red Hat OpenShift, DigitalOcean and many more.

The list of essential design concepts for designing robust, scalable, and secure cloud-based systems is present on Table 2. Each principle identifies the specific problem it addresses.

Table 2. List of key design principles.

Name	Description
Separation of Concerns	A design guideline for dividing distinct sections of a computer program. Each module and object must have its own purpose and context. Accordingly, there are more opportunities for module development, reuse, and autonomy.
Encapsulation	A way to restrict direct access to certain segments of an element so that people could not view the state values of all of an object's variables. Encapsulation can be used to cover up both the data members and the data functions or methods.
Single Responsibility	The principle stating that "A module should be responsible to one, and only one, actor." (R. C. Martin, 2017). To put it another way, each piece in the design must have a single purpose. It is closely related to the concepts of coupling and cohesion.
Dependency Inversion	Research by R. C. Martin shows that this principle is a specific way to connect software modules in a loose way. In accordance with this approach, the typical dependence connections between high-level, policy-setting modules and low-level, dependency modules are reversed, making high-level modules independent of the implementation details of low-level modules.
"You Are Not Going to Need It" (YAGNI)	A fundamental principle of extreme programming (Newkirk & Martin, 2001). YAGNI says, "Do not add

	functionality unless it is considered required." In other words, create the code required for the given circumstance. Must not add anything that is unneeded. For the time being, adding logic to the code should not take into account what may be required in the future.
"Keep It Short and Simple." (KISS)	This idea relates to the simplification of functionality implementation. Less complicated code is easier to read and hence easier to maintain.
Factory	This is one of the well-known Gang of Four design patterns (1994). Its purpose is to create an interface for object creation while allowing subclasses to choose which class to instantiate. It's also known as a virtual constructor.

2. The Domain-Driven Design features in the context of cloud services

A web service, whether a monolith or a microservice, has certain features, the most important of which are the volume of data handled, performance requirements, business logic, and technological complexity. DDD approaches are useful for projects with a large number of complex business rules where it could solve the complexity of business logic. In other words, the main goal of DDD ideas is to address the complexity of business logic.

The classic approach, as described by T. Erl in his book "SOA Principles of Service Design" (2023) incorporates the separation of services based on of their technical and functional characteristics. E. Evans (2003), on the other hand, says that DDD gives the key ideas needed to separate web services into different parts. The DDD approach provides a means of representing the real world in the architecture, for instance, by using bounded contexts to represent organizational units and also identifying and focusing on the core domain. These characteristics lead to improved software architecture quality (E. Landre, 2016).

The focus should always be on the core domain. Business logic complexity is the first indicator of how complicated the problem domain in which a software works is. For example, a CRUD application that performs basic create, read, update, and delete operations doesn't carry a lot of complexity with it. This case can be managed with simpler approaches. At the same time, an order management system, which automates a big chunk of the company's activity, must model all the processes the company acts upon and thus handle a lot of complex business roles. The business logic complexity of such a system may be extremely high. Another attribute is

the technical complexity, which is the number of algorithms that need to be implemented to make the software work.

In the book "Patterns of Enterprise Application Architecture", Martin Fowler (2012b) presents a diagram with time and cost on the Y axis and complexity on the X axis. In accordance with data-centric design patterns, the curve indicates that beyond a certain level of complexity, even a small increase in complexity results in a significant cost peak.

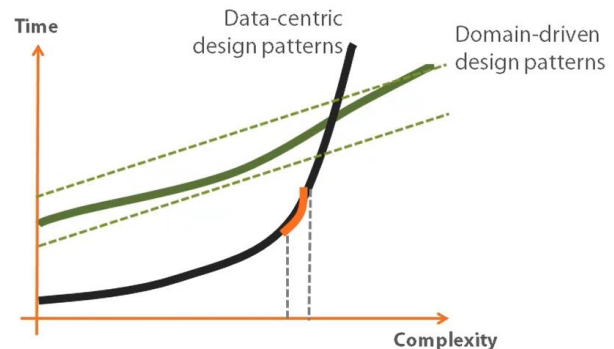


Figure 1. Time and complexity diagram. (Adapted from Martin Fowler's book "Patterns of Enterprise Application Architecture", 2012b)

On the other hand, the time and cost of a project designed from a domain-centric perspective tended to increase linearly with complexity, but the startup costs were quite high. Domain-driven design (DDD) says that use cases should be modeled based on how the business actually works. In the context of building applications, DDD talks about problems as "domains" (César de la Torre, 2022). DDD calls separate problem areas "bounded contexts" and stresses the need to talk about these problems in the same way. DDD suggests many technical ideas and patterns to help with the internal implementation. These include domain entities with rich models (no "anemic" domain models), value objects, aggregates, and aggregate root rules. Ubiquitous language, bounded context, and core domain are the strategic elements and the most important parts of DDD. The other ideas, such as entities, value objects, and repositories, are the steps for building a software project.

Some individuals view these technical rules and patterns as difficult-to-learn obstacles that make it challenging to employ DDD methodologies. However, the most critical aspect is arranging the code so that it is matched with the business problems. (Bill Wagner, 2022).

Some important features in this context according to us are ubiquitous language, bounded contexts, entities, value objects, aggregates, repository and domain events. They are relevant in cloud services too and below we shortly outline each one.

Core principles of DDD make it easier for domain experts and software engineers to talk to each other by defining an explicit ubiquitous (universal) language. This language assists in bringing together the stakeholder, the designer, and the programmer so that they may construct the domain model(s) and then put them into action (Hippchen, Benjamin, 2017). Code written in the ubiquitous language can provide a hint for some edge cases that weren't clear enough at the start, or it can rewrite the problem statement in a much cleaner and more concise manner. For the idea of a ubiquitous language to work, the code base needs to be in sync with the terminology, or, more specifically, classes and tables in the database need to be named after the terms in the ubiquitous language. Common nomenclature facilitates understanding of user requirements. Batista's research indicates this helps bridge the gap and establishes the foundation for effective communication (2022). It seeks to develop a standard, business-oriented language. The basic objective of the language is to prevent misunderstandings and incorrect assumptions.

The bounded context is the small area within the domain that gives each element of the ubiquitous language its own meaning (Merson & Yoder, 2020). Quite often, an application's code base becomes unmanageable as its volume increased. Code elements that make sense in one part of the system may seem completely irrelevant in another. In this case, the best solution would be to explicitly separate these parts from each other. A bounded context illustrates how the program and its development were structured. Frequently, it corresponds to a subdomain, which indicates how the business or domain activity is divided (Khononov, 2021).

Even though a DDD application is behavior-driven, objects are still necessary. DDD expresses two types of objects: those that are defined by an identity and those that are defined by their values. An entity is something that can be tracked, located, retrieved, and stored by an identity key (Thalheim, 2010). Because they serve such a vital role in the system, entities acquire a tremendous amount of functionality. Applying the single responsibility pattern to entities is indeed a nice idea. Anything that doesn't fit that description should be placed elsewhere. Instead of being defined by their attributes, entities are things that were defined by a thread of continuity and identity (Evans, 2003).

Martin Fowler's definition of a Value Object (2016) is as follows: A small simple object, whose equality isn't based on identity. It is an item that is used to quantify, measure, or characterize a certain topic. Because the property values define it, it ought to be immutable. Value objects may have methods and behavior, but they should never have side effects.

Vaughn Vernon says in his book *Implementing Domain-Driven Design* that value objects should be used instead of entities whenever possible. Even if a domain notion must be treated as an entity, the entity should indeed be designed to contain values rather than other entities.

According to Eric Evans, an aggregate is a collection of connected items that are modified as a single entity. Aggregates consist of one or more entities and value objects that change together. Aggregates are treated as a unit for data changes. Before making modifications, it is necessary to evaluate the consistency of the whole aggregate. Every aggregate must have an aggregate root, which is the parent object of all members of the aggregate. In some cases, the aggregate may have rules that make sure all of the objects' data is consistent. Data changes in aggregate should adhere to ACID, which means they should be atomic, consistent, isolated, and long-lasting (Jovanovic & Benson, 2013). It is also the responsibility of the aggregate root to maintain its invariants, such as the number and type of its components. A condition that must always be true for the system to be in a consistent state is an invariant.

A repository is a collection of items of a particular type that may be queried similarly to a collection, but with extra options. Repositories offer a unified abstraction for all persistence-related problems (Bahri & Williams, 2022). This makes it easy for clients to get model objects and manage their lifecycle. The public interface of a repository communicates design decisions very clearly. Only a few things should be directly accessible, therefore repositories give and regulate this access. An important benefit is that repositories make the code easier to test. They reduce the tight coupling with external resources like as databases and data providers, which would traditionally make unit testing challenging. When code for data access is wrapped in one or more well-known classes, it is easier and safer to use (Gorman, 2021).

As objects, domain events are an integral component of a bounded context. They give a way to talk about important things that happen or change in the system. Then, loosely connected parts of the domain can respond to these events (Garverick & McIver, 2023). In this manner, the objects that raise the events do not need to consider the action that must occur when the event occurs. Similarly, event-handling objects do not need to know where the event originated.

Vaughn Vernon describes domain events, saying they should be used to capture an occurrence of something that happened in the domain. They should be part of the ubiquitous language. Events are helpful because they signal that a certain thing has happened.

A domain event is essentially a message, a record of something that happened in the past.

3. Managing the complexity issues in cloud services by layers approach

The most important aspect of designing and establishing a service is setting its boundaries. DDD patterns assist in the understanding of the domain's complexity. Each bounded context identifies the entities and value objects, characterizes them, and combines them into a model of the domains. Choosing where to draw the border between bounded contexts requires balancing two competing objectives. Creating a barrier around items that need cohesion is the first step. The second goal is to avoid chatty inter-unit communications. These objectives may conflict with one another. Balance should be accomplished by decomposing the system into the smallest units feasible (Zimarev, 2019b). In a single-bound context, cohesion is crucial. Another way to look at this aspect is autonomy. A unit is not completely autonomous if it relies on another unit to fulfill a request directly.

DDD concepts create a structure known as "onion architecture". It is called an onion because it has numerous layers and a central core. The top layers are dependent on the lower layers, yet the lower layers have no knowledge of the upper. Onion architecture emphasizes the fact that the core elements of the domain model should act in isolation from each other.

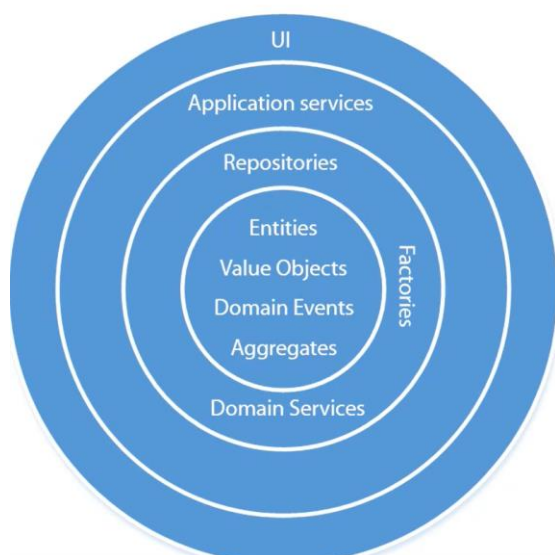


Figure 2. Building blocks of domain-driven design in onion architecture (Adapted from book "Domain-Driven Design: Tackling Complexity in the Heart of Software", 2003)

The core part of this so-called onion is the notion of entity, value object, domain event, and aggregate.

The next layer consists of repositories, factories, and domain services. Application services go beyond that. The code working with the data storage must be gathered under the repositories in the domain model. These four elements: entities, value objects, domain events, and aggregates, are the most basic. They can refer to each other. For example, a value object can keep a reference to an aggregate root, but cannot work with other DDD notions, such as repositories and factories. Similarly, repositories, factories, and domain services can know about each other and the four basic elements, but they should not refer to the application services. The main reason for the isolation is the separation of concerns.

Entities, value objects, domain events, and aggregates carry the most important part of the application, its business logic. Repositories and factories can keep some of the business logic as well.

It is crucial to leave entities and value objects to do only one thing: represent the domain logic in the application. In practice, it means they shouldn't contain any knowledge about how they are preserved or created. These two operations must be up to the standards of repositories and factories. They also shouldn't contain any knowledge about the tables and columns in the database where they are stored. This must be given away to database members. All they should know is the domain they represent.

Layers are strict areas of concern. Each one may only communicate with peers above or below. High abstraction and isolation, structured communication, and ease of scaling out are some advantages. Deep call chains have the disadvantage of hiding complexity, reducing performance, and requiring the lowest layer to cover all use cases.

Most enterprise applications with significant business and technical complexity are defined by multiple layers. (Mike Rousos, 2022). The layers are a logical artifact that has nothing to do with how the service is deployed. They exist to help developers manage the complexity of the code. Different layers may have different types, necessitating translations between them. A domain model must be controlled by aggregate roots that ensure that all invariants and rules related to that group of entities are performed through a single entry-point or gate (Bill Wagner, 2022).

Domain model layer is in charge of representing business concepts, business situation information, and business rules (Eric Evans). The domain model layer is where the business is expressed; it is the heart of business software. According to the Persistence Ignorance (todo) and Infrastructure Ignorance (OREN EINI, 2008) principles, this layer must not know anything about how data is stored. The infrastructure layer should be in charge of these persistence tasks. Domain entities shouldn't directly

depend on any data access infrastructure framework, like by inheriting from a base class. Even though the persistence ignorance principle is important for the domain model, concerns about persistence should not be ignored. It is still important to understand the physical data model and how it maps to the entity object model; otherwise, impossible designs would be created (César de la Torre, 2022).

The **application layer** defines the functions of the software and directs the expressive domain objects to solve problems. This layer is in charge of tasks that are important to the business or are needed to work with the application layers of other systems. This layer is kept thin. It contains no business rules or knowledge, but only coordinates tasks and delegates work to domain object collaborations in the next layer down. It does not have a state reflecting the business situation, but it can have a state that reflects the progress of a task for the user or the program (Eric Evans). It gives the task of executing business rules to the domain model classes (aggregate roots and domain entities), which will then update the data in those domain entities. The goal is for the presentation and application layers to have nothing to do with the domain logic in the domain model layer, its invariants, the data model, or any business rules that go with it (Bill Wagner, 2022).

The **infrastructure layer** is where data from domain entities is stored in databases or another persistent store. An example is using object-relational mapping framework code to implement the repository pattern classes. According to the persistence and infrastructure ignorance principles, the infrastructure layer must not "contaminate" the domain model layer. The entity classes in the domain model must be isolated from the infrastructure.

Dependencies between the three layers mentioned above is shown on Fig.3. The application layer depends on domain and infrastructure, and infrastructure depends on domain, but domain does not depend on any layer.

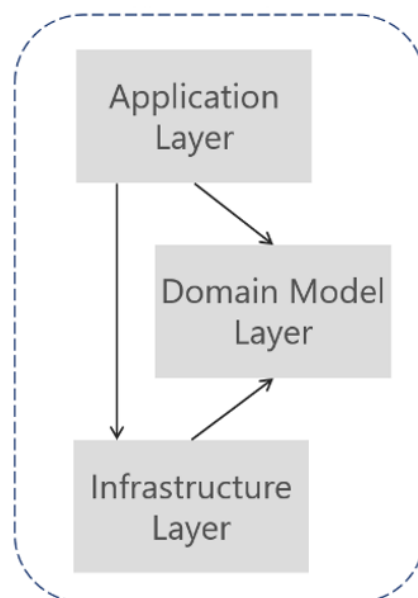


Figure 3. Dependencies between layers in DDD Authors: Cesar de la Torre, Bill Wagner

The DDD patterns presented in this article should not be applied universally. They introduce constraints, which provide benefits such as higher quality over time, especially in commands and other code that modifies system state. However, those constraints add complexity with fewer benefits for reading and querying data (César de la Torre at all, 2022). Example: Because of how they work together in the domain, the aggregate pattern treats many domain objects as a single unit. If you treat multiple objects as a single aggregate. There is no advantage for read-only queries, but it may raise the complexity of response logic.

4. Using command and query responsibility segregation and event-sourcing in cloud services

Command and Query Responsibility Segregation (CQRS) was introduced by Greg Young back in 2010. Greg based this idea on Bertrand Meyer's command-query separation principle. **CQS (?)** for short, states that every method must either be a command that executes an operation that modifies the state of the system, or a query that provides data to the caller, but not both. So, asking a question shouldn't affect the outcome of the response. Methods should only return a value if they are referentially transparent and don't have any side effects, like changing the state of an object or a file in the file system, etc. To follow this principle, if a method changes some piece of state, this method should always be of type void otherwise, it should return something. This increases the readability of the code base. However, it is not always practical to stick to the command-query separation paradigm; there are occasions when it makes more sense for a

method to have both a side effect and a return value. An example here is Stack. Its Pop method removes the element pushed into the stack last and returns it to the caller. This solution violates the CQS concept, yet separating these duties into two distinct functions is illogical.

The relationship between CQS and CQRS is that CQRS extends the same notion to a higher level and is seen as an architectural pattern. Instead of focusing on methods like CQS, CQRS applies the same principles to the model and its classes. CQRS facilitates the separation of a single, unified domain model into two distinct: one for command management, or writes, and the other for query processing, or reads. CQRS facilitates the separation of a single, unified domain model into two distinct models: one for managing commands, or writes, and the other for processing queries, or reads (Fowler, 2011). Similarly, CQS encourages splitting a method into two parts, a command and a query. In many cases, CQRS is related to more advanced scenarios (César de la Torre, 2022).

Typically, it is quite difficult to create a unified model since the command side and the query side have very distinct needs. By concentrating on each case individually, a different strategy that makes the most sense may be developed. In the end, there are two models, each of which specializes at a certain purpose. The separation aspect is achieved by grouping query activities into one layer and commands into another. Each layer has a unique data model. More importantly, the two layers can be within the same tier, or they could be implemented on different microservices or processes so they can be optimized and scaled out separately without affecting one another (Bill Wagner, 2022). This can be seen as the single responsibility principle being used at the architectural level.

The CAP theorem and CQRS have a close relationship. A distributed data store cannot ensure more than two of consistency, availability, and partition tolerance simultaneously, according to the CAP theorem (Brewer, 2012). If consistency is maintained, every "read" operation returns the most recent "write" or an error. Availability, on the other hand, implies that every request receives a response, even if all system nodes are down. With partition tolerance, the system continues to function even when communications are lost or delayed across network nodes. Due to the impossibility of selecting all three, it is crucial to find a compromise. CQRS is strong due to the number of possibilities it provides.

In summary, CQRS focuses on making decisions that are optimal for various circumstances. For the command and query sides, multiple levels of consistency could be selected. CQRS is frequently

referred to as an intermediate phase before event sourcing.

Event sourcing is a design technique based on the concept that all changes to the state of an application throughout its lifetime are recorded as a series of events. As a result, serialized events become the fundamental building blocks of the application (Dominguez et al., 2012). In event sourcing approach the programs store transactions but not their respective states. When a state is needed, all transactions from the beginning of time are applied (R. Martin, 2017b). Nothing is deleted or updated from the data repository. Because of it, there cannot be any concurrent updating issues. Most applications work by storing the current state of domain entities and starting business transactions from this state. Instead of storing all the information in the columns of a single record or in the properties of a single object, the state of the entities is described by the sequence of events that led to it having a given list of items. This is an event-based representation of an entity. As described previously in the article, an "event" is something that occurred in the past and is an expression of the ubiquitous language.

Storage may be relational, document-based, or graph-based, therefore events might be stored in a NoSQL database, an ad-hoc relational table, or using a specific solution such as Azure Cosmos DB, FaunaDB, and many more (Blokdyk, 2022). Any kind of event storage is append-only and does not allow deletions. To obtain the entire state, it is necessary to replay the program timeline from the beginning. Using recorded events, it is possible to reconstruct the state of an aggregate. This may sometimes need the management of massive volumes of data. In this case, snapshots, which represent the state of the entity at a certain point in time, may be specified (Baptista & Abbruzzese, 2022b). Once stored, events are immutable. It is possible to duplicate and repeat events for scalability reasons.

Replay involves examining this data and using logic to retrieve relevant information. Other, more intriguing situations, such as business intelligence and statistical analysis, may be addressed by ad hoc projections.

All the aforementioned patterns, techniques, and principles are geared toward the design and development of simple, intuitive, flexible, testable, and maintainable cloud software architecture. A software architecture is a collection of patterns that may stack inside one another securely. The "Clean architecture" is a philosophy of architectural essentialism and mainly a cost-benefit argument. Users' use cases and mental models need to be reflected in the system. And that is what clean architecture focuses on. It builds only what is necessary, when it is necessary and optimizes for

maintainability. The topic is also connected to the notion of "clean code." Clean code reads like well-written prose. It never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control (Booch et al., 2007).

It's important to emphasize that CQRS and most DDD patterns (like DDD layers or a domain model with aggregates) are not architectural styles but only architecture patterns. Microservices and SOA are examples of architectural styles. CQRS and DDD patterns describe something inside a single system or component (Bill Wagner, 2022). At an architecture pattern level, the design of each bound context in that application shows its own trade-offs and internal design decisions.

5. Applying test-driven development practice in cloud services

Test-Driven Development is a software practice in which a failed test is built before any production code is written and is then used to influence the design of the architecture. There is a three-step procedure known as "red, green, and refactor". Creating a failing test for a piece of functionality is the initial red step. The second phase is the "green step," during which sufficient production code is created to make the failed test pass. Refactoring is the last phase in which both test and production code are enhanced to maintain high quality (Myers, 2022). This cycle is repeated for each piece of functionality in order of increasing complexity in each method and class until the whole feature is finished. By using TDD, the testing process is what guides the design. Testable code is what produces maintainable code (Beck, 2002b).

In the field of software testing, there are several different sorts of tests. Some tests are subject matter based. For example, unit, integration, component service, and user interface testing. Some are determined by the purpose of the test. For example, functional tests, acceptance tests, smoke tests, and exploratory testing. Others, though, are determined by how they are being tested: automated, semi-automated and manual tests.

The test automation pyramid (Fig. 4) was first described by Mike Cohn in his book *Succeeding with Agile: Software Development Using Scrum*. The test automation pyramid depicts the types of automated tests that should be performed at various stages of the software development lifecycle and how often they should occur in a testing suite to ensure the quality of the program. The notion behind the pyramid is that testers should dedicate more effort to basic tests before moving on to more complicated ones.

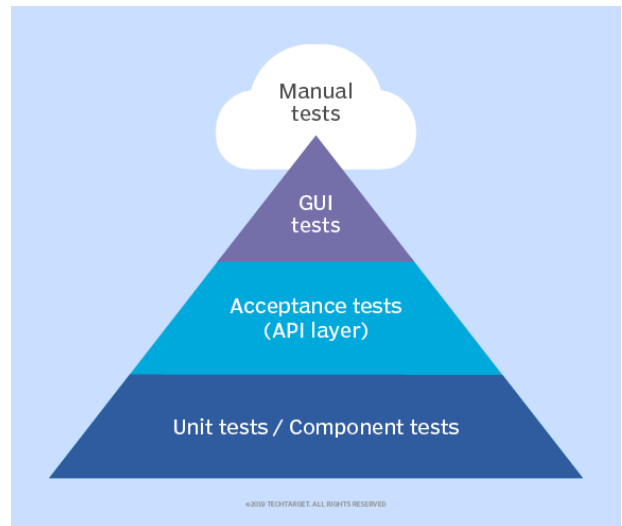


Figure 4. The agile test automation pyramid was introduced by Mike Cohn in his book *Succeeding with Agile*.

On fig. 4 four different test kinds are identified:

- 1) Unit tests - automated tests that check how well a single piece of code works on its own;
- 2) service tests - automated tests that check how well a group of classes and methods that provide a service to users work;
- 3) UI tests - automated tests that check that the whole application works (from the user interface to the database);
- 4) Manual tests - tests done by a person, also check the full application's functionality;

The test automation pyramid captures the essence of how each type of test becomes more expensive. As a result, the system should have many low-cost tests and a small number of high-cost tests.

6. Conclusion

The cleaner the domain model is kept, the easier it is to reason about it and to extend it later on. Inability to maintain proper separation of concerns in enterprise-level applications is one of the biggest reasons why code bases become a mess, which leads to delays and even failure of the project. It is not always possible to separate them completely, though, and there will always be some elements not related to the domain. Nevertheless, it is possible to keep those elements under control so that they introduce almost no overhead to the domain clauses. As this article focuses mostly on the foundations, a case study on the domain-driven software development process might be presented as a continuation.

Acknowledgements

This research is financially supported by NPD-XX/2023 from University of Economics - Varna Science Fund.

References

- [1]. Wong, B., & Kokko, H. (2005). Is science as global as we think?. *Trends in ecology & evolution*, 20(9), 475-476.
- [2]. Haykin, S. (1994). *Neural networks: a comprehensive foundation*. Prentice Hall PTR.
- [3]. Hennessy, J. L., & Patterson, D. A. (2012). *Computer architecture: a quantitative approach*. Elsevier.
- [4]. FESTO. (2019). Fluidic Muscle DMSP/MAS. Retrieved from:
https://www.festo.com/rep/en_corp/assets/pdf/info_501_en.pdf [accessed: 19 September 2021].