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* – With the proliferation of cloud-native services, the need for efficient software design strategies has become of the utmost importance. 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. It emphasizes multiple techniques for expressing complex business logic by facilitating greater scalability, flexibility, and maintainability. The significance of the system's availability, reliability, and resilience may prevent an organization from failure and support its growth. This article explores the essential components of Domain-Driven Design (DDD), their integration with cloud-native technologies, and the benefits and challenges associated with them. In addition, the research seeks to contribute to the expanding corpus of knowledge in this area and to assist software architects and developers in utilizing design principles.**

***Keywords* – domain-driven design, cloud-native services, scalability, maintainability, modularity, distributed systems, software architecture.**

1. Introduction

The emergence of cloud-native services has revolutionized the development and deployment of software systems. These services take advantage of the agility, adaptability, and a fault tolerance that cloud platforms offer. However, designing cloud-native applications present unique challenges associated with distributed systems, microservices.

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: -----.

Domain-Driven Design (DDD) is a software development methodology that emphasizes the application domain, its concepts, and their relationships as the motivating factors for architecture design [37], [38]. The business perspective of domain is: "A field | industry in which the business operate, composed of multiple sub-domains". Each sub-domain has a different level of complexity and could be categorized into 3 types: Generic, Core, Supporting. Businesses invest in software to meet specific requirements or to address particular issues. For a complete understanding of the problem, architects must first comprehend the domain! Core principles of DDD include capturing relevant domain knowledge in domain models, which can include both structural and behavioural aspects, collaborative modelling between domain experts and software engineers. Domain-driven design provides patterns, activities, and examples of how to build a domain model, which is its main artifact [2]. This article analyses the potential of DDD as a governing principle for designing cloud-native services, with the goal of optimizing the development processes.

The list of essential design concepts for designing robust, scalable, and secure cloud-based systems is present on Table 1. Each principle may be used as a solution to a commonly occurring problem.

Table 1. 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 basic concept asserting that "A module should only be accountable to a single actor." [40]. 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 [29], [30] 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 [29], [30]. 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 [23]. 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.

When creating a cloud solution, one of the first decisions to make is which service(s) to utilize in order to operate the applications [33]. Table 2 shows the choices for which cloud services are best for which types of applications.

Table 2. 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 hosting services that cover the complexity of the operating system and infrastructure while hosting an application [10]. They are highly available by default and will be operational at least 99.95 percent of the time. They share potent characteristics such as automatic scaling, zero-downtime deployments, and straightforward authentication and authorization [9]. Some of them enable debugging the application while it is in production, using tools such as Snapshot Debugger [33].

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 example, there is an offline sync that empowers the mobile app to keep functioning if there's no connection to the backend, and the sync is refreshed whenever the connection is re-established [21]. Another feature is sending push notifications to the mobile apps, regardless of the platform they run on (iOS, Android, or Windows).

Serverless 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" [24]. Even scaling is handled by these functions. Cloud providers transparently spawn additional functions to handle heavy loads, and they disappear after the code has completed executing. As a function, companies only pay for the software that is executed, and not for a service that is always running and waiting to be triggered.

Existing applications could be lifted and relocated from virtual machines (VM) operating in a local datacentre 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. Even so, running the application in a virtual machine doesn't offer some optimizations. The operation staff is also accountable for maintaining the operating system and anti-virus software [9]. 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 [41]. 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 [36]. 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 [42]. Such cloud solutions are Azure Kubernetes Service, Amazon EC2 & EKS, Google Kubernetes Engine, Red Hat OpenShift, DigitalOcean and many more. Microservices architecture is a catalyst and enabler for continuous business transformation.

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

A web service, whether a monolith or part of distributed system, 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 domain logic, which are the business rules, validations and calculations.

The classic approach, as described by T. Erl in his book "SOA Principles of Service Design" [13] incorporates the separation of services based on of their technical and functional characteristics. It focuses on core capabilities exposed as services. E. Evans [14], [15], 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 a structured representation of a solution that meets the requirements in the problem space. These characteristics lead to improved software architecture quality [25].

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. A CRUD application that needs to perform fundamental create, read, update, and delete operations, is not particularly complex. This situation can be handled with less complicated methods. Simultaneously, an order management system [31], which automates a significant portion of a company's activity, must model all the processes upon which the company acts and therefore manage a large number of complex business responsibilities. This system's business logic complexity may be extremely high [9].

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” [20], Martin Fowler 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.

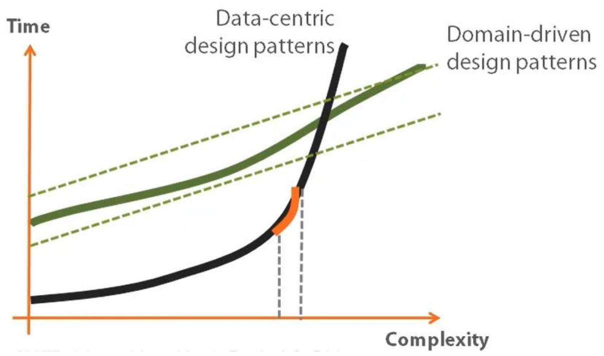

Figure 1. Time and complexity diagram. (Adapted from Martin Fowler's book " Patterns of Enterprise Application Architecture")

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 start-up costs were quite high. DDD says that use cases should be modelled based on how the business actually works.

DDD offers a variety of technical concepts and patterns to assist in the internal implementation. 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, aggregates, 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 [47].

Each industry | profession has its own lingo. For building complex systems IT Teams must learn the business terminology used by the expert & stakeholders. A core principle of DDD make easier for domain experts and software engineers to talk to each other by defining an explicit ubiquitous (universal) language (UL). 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 [3]. Code written in the UL 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 [48]. For the idea of a UL 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 [3] indicates this helps bridge the gap and establishes the foundation for effective communication. It seeks to develop a standard, business-oriented language. The basic objective of the language is to prevent misunderstandings and incorrect assumptions. UL is utilized in: documentation, conversations, domain experts, delivery teams, app code, testing code. UL evolves over a period and may be managed on any knowledge collaboration platform. UL helps in identifying focus areas for knowledge crunching, which is the process of "coping" the knowledge received from the experts into domain models.

The bounded context (BC) is a small area within the domain that gives each element of the ubiquitous language its own meaning [31]. Quite often, an application's code base becomes unmanageable as its volume increased. Elements of code that make sense in one portion of the system may appear irrelevant in another. In this situation, the optimal solution would be to explicitly separate these components []. 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 [24]. Each BC is represented with its own domain, and it is developed independently. Domain model built for a BC is applicable only within its boundaries.

Even though a DDD application is behaviour-driven, objects are still necessary. DDD conveys several distinct kinds of objects, defined by their identities or values [39]. Table highlighting them.

Table 3. List of DDD core concepts and principles.

Name	Description
Entity	Represents a uniquely identifiable business object that encapsulates attributes and a well-defined domain behaviour. Definition of entity consist of attributes and behaviour. It is something that can be tracked, located, retrieved, and persisted in long term storage.
Value Objects	A small simple object, whose equality isn't based on identity [18],[19]. It is an item that is used to quantify, measure, or characterize a certain topic. Value objects may have methods and behaviour, but they should never have side effects. In his book Vaughn Vernon [43] says that value objects should be used instead of entities if possible.
Aggregate	Collection of connected items that are modified as a single entity [16], [17]. Aggregates are treated as a unit for data changes. They consist of one or more entities and value objects that change together. 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. 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). For creating complex aggregates, the factory pattern can be used.
Repository	A collection of items of a particular type. Repositories offer a unified abstraction for all persistence-related problems [34]. This makes it easy for clients to get and manage model objects. 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).

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 and combines them. 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 [49]. 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 fulfil 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 that the core elements of the domain model should act in isolation from each other.

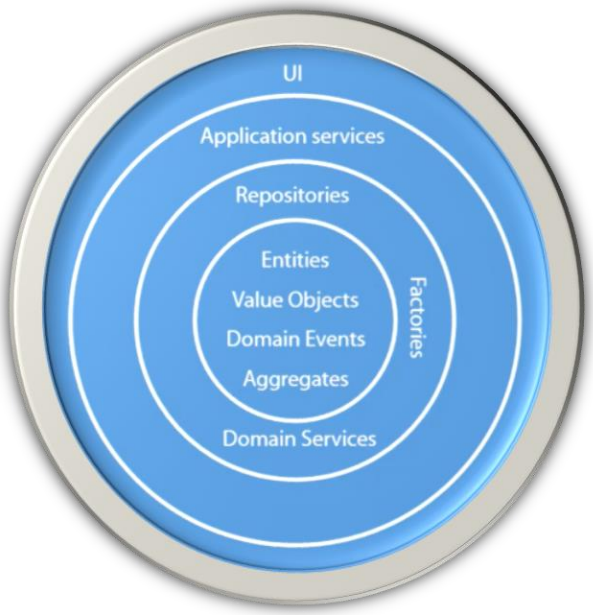

Figure 2. Building blocks of domain-driven design in onion architecture (Adapted from book "Domain-Driven Design: Tackling Complexity in the Heart of Software", 2003)

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.

Most enterprise applications with significant business and technical complexity are defined by multiple layers [13]. These layers are a logical artifact that help developers manage the complexity in the code. They have nothing to do with how the service is deployed. Depending on the specific implementation, the elements can be organized differently when DDD principles are applied. Nonetheless, as shown in Figure 3, there are a few common layers.

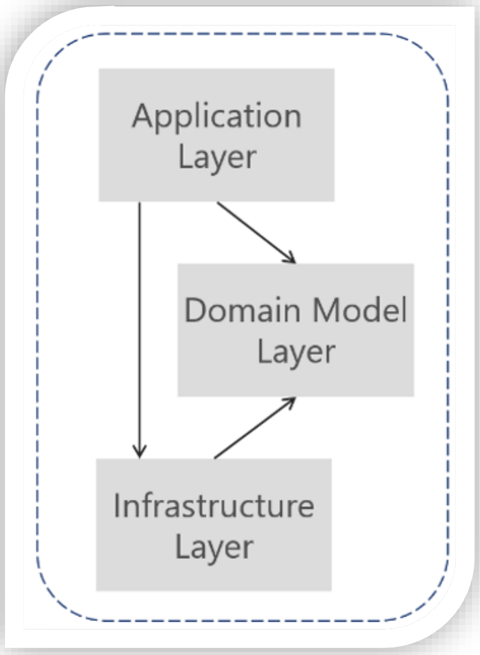

Figure 3. Dependencies between layers in DDD Authors: Cesar de la Torre, Bill Wagner

The **application layer** coordinates the execution flow between various domain objects/entities to solve problems. It specifies the use cases and operations that can be carried out within the service. It orchestrates interaction between the UI and the core elements. Commonly, the application layer is implemented as a web API project, which implements the interaction, remote network access, etc. The application layer depends on domain and infrastructure.

Domain model layer encapsulates the business logic and principles and constitutes the core of the service. It contains domain objects/entities, aggregates, value objects, and domain services. The domain layer concentrates on solving business problems and expresses the business domain's concepts and behaviours. Should have only the domain code, just completely decoupled POCO classes, implementing “the heart of the software”. The domain layer does not depend on any other.

The **infrastructure layer** provides the service with technical capabilities and support. It comprises data access objects (DAOs), repositories, integrations with external services, messaging frameworks, cache mechanisms, and other infrastructure-related implementations. The infrastructure layer communicates with external systems and services to provide persistence, communication, and other infrastructure-related concerns. Infrastructure layer depends on domain layer.

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).

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

CQRS, acronym for Command and Query Responsibility Segregation, was introduced by Greg Young back in 2010. Greg based this idea on Bertrand Meyer's command-query separation principle. Command-query separation (CQS), 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 is the linear data structure “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 by facilitating 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 is an object-oriented expression of the domain and is frequently associated with more complex business contexts (César de la Torre, 2022).

Typically, it is quite difficult to create one specific unified model since the command and the query sides 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. The application layer turns any input into a command or a query and sends it to a shared communication channel (message handler). Commands, queries, and events are three categories of messages in an application. They are all part of the core domain model, located in the centre of the onion architecture. Commands are telling the application to do something, queries are asking it about something, and events are the informational messages. Commands trigger a reaction in the domain model, while events are the result of that reaction. Naming guidelines are associated with all three types of messages, with commands always being in the imperative tense, queries usually starting with the word Get, and events always being in the past tense. It is important to use the ubiquitous language.

In addition, the query and command handlers can be implemented within the same tier or on distinct services so that they can be optimized and scaled independently without affecting one another, offloading, as well, the complexity from the code base (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. The CAP theorem states that a distributed data store cannot simultaneously guarantee more than two of following tree: consistency, availability, and partition tolerance. [7]. 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 choosing all three options, it is necessary to reach a compromise. CQRS is powerful because it offers numerous opportunities by focuses on making decisions that are optimal for various circumstances.

By adopting CQRS, developers can design cloud-native services that efficiently handle high query loads while ensuring data consistency through strict command processing. CQRS is commonly referred to as an interim stage preceding event sourcing. Event Sourcing complements CQRS by capturing all changes to the system's state as a sequence of events.

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. 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 [29],[30]. Nothing is deleted or updated from the data repository. Because of this fact, 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. 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.

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 [21]. 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 [44] 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.

Event storage may be relational, document-based, or graph-based, therefore events might be stored in a SQL or NoSQL database or using a specific solution such as “RavenDB” or “FaunaDB” [4]. Table 3 describes some cloud-based options.

Table 3. Cloud data-storages' suitability for various business cases

	Relational	Unstructured	Semi Structured	Tunable Consistency	Geo-Replication	Large Data
Azure SQL	✓				✓	
Azure Cosmos		✓	✓	✓	✓	✓
Azure Blob		✓			✓	✓
Amazon RDS	✓				✓	✓
Amazon Dynamo		✓	✓	✓	✓	✓
Amazon S3		✓			✓	✓
Google SQL	✓				✓	✓
Google Firestore		✓	✓	✓	✓	✓

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 algorithm involves examining the data and using logic to retrieve the relevant information. Other, more intriguing situations, such as tracking the history of a resource, business intelligence and statistical analysis, may be addressed by ad-hoc projections. Events, as well, provide a powerful and efficient approach to data warehousing, supporting by cloud services like Amazon Redshift, Google BigQuery and Azure Synapse Analytics.

5. Applying test-driven development practice in cloud services

Test-driven development (TDD) and DDD are two potent methodologies that, when combined, can increase the quality of cloud services and the development process. By employing these practices, developers and quality assurance engineers can create a system that is more robust and reliable. TDD encourages a rigorous testing process in which tests are written prior to the implementation code, which follows the best practices, ensuring that the intended functionality is met. 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 [32]. 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 [5].

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 [1]. 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.

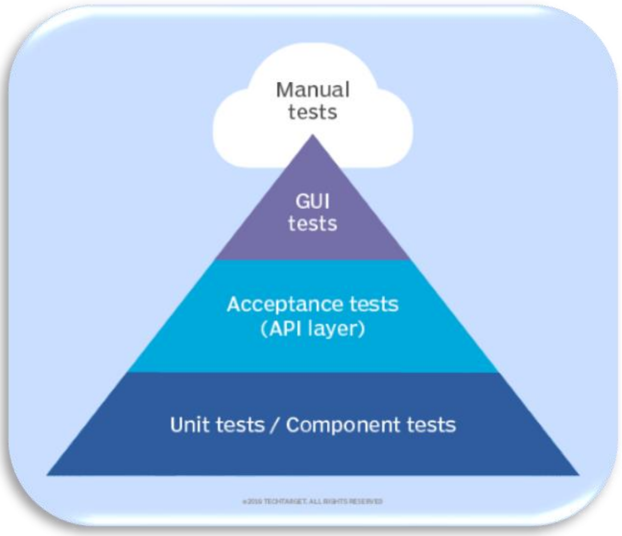
Manual tests
GUI tests
Acceptance tests (API layer)
Unit tests / Component tests
©2016 TECHTARGET. ALL RIGHTS RESERVED

Figure 4. The agile test automation pyramid by Mike Cohn (Succeeding with Agile: Software Development Using Scrum).

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.

By implementing TDD, programmers could identify potential problems early on and validate the veracity of the domain models. In addition, the iterative nature of TDD enables frequent feedback, which facilitates continuous refinement and adaptability in cloud service development [37].

Conclusion

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.

In conclusion, Domain-Driven Design (DDD) approaches have emerged as a valuable methodology for building cloud-native services architecture. By focusing on the core business domain and encapsulating it in a well-defined bounded context, DDD helps to create modular, scalable, and maintainable systems. The cloud-native approach complements DDD by leveraging modern cloud technologies and design patterns to improve scalability, resilience, and efficiency. By combining these two approaches, organizations can build systems that are not only technically robust but also aligned with their business goals, requirements, and objectives. Ultimately, the adoption of DDD and cloud-native architectures can help organizations to innovate faster, reduce costs, and deliver better value to their customers, as well as to stay competitive in a rapidly changing digital landscape.

The cleaner the domain model is kept, the easier it is to extend it later. 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. 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]. Ashbacher, C. (2010). *Succeeding With Agile: Software Development Using Scrum, by Mike Cohn*. *The Journal of Object Technology*, *9*(4). https://doi.org/10.5381/jot.2010.9.4.r1.

[2]. Avram, A. (2007). *Domain-Driven Design Quickly*. Lulu.com.

[3]. Batista, F. (2019). *Developing the ubiquitous language*. Retrieved from: https://thedomaindrivendesign.io/developing-the-ubiquitous-language [accessed: 19 May 2023].

[4]. Betts, D., Dominguez, J., Melnik, G., Simonazzi, F., & Subramanian, M. (2012). *Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure*.

[5]. Bissi, W., Neto, A. T., & Emer, M. C. F. P. (2016). *The effects of test driven development on internal quality, external quality and productivity: A systematic review. Information & Software Technology*, *74*, 45–54. https://doi.org/10.1016/j.infsof.2016.02.004.

[6]. Booch, G., Maksimchuk, R., Engle, M., Conallen, J., Houston, K., & Young, B., PhD. (2007). *Object-Oriented Analysis and Design with Applications*. Pearson Education.

[7]. Braun, S., Bieniusa, A., & Elberzhager, F. (2021). *Advanced Domain-Driven Design for Consistency in Distributed Data-Intensive Systems*. *European Conference on Computer Systems*. https://doi.org/10.1145/3447865.3457969

[8]. Brewer, E. (2012). *Pushing the CAP: Strategies for Consistency and Availability*. *IEEE Computer*, *45*(2), 23–29. http://dx.doi.org/10.1109/MC.2012.37

[9]. Caron, R. (2023). *Get the Azure Quick Start Guide for .NET Developers*. Microsoft. https://azure.microsoft.com/en-us/blog/get-the-azure-quick-start-guide-for-net-developers/

[10]. De La Torre, C. (2022, September 09). *Containerized Docker Application Lifecycle with Microsoft Platform and Tools*. Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/architecture/containerized-lifecycle/

[11]. De La Torre, C., Wagner, B., Rousos, M., (2023, March 22). *.NET Microservices. Architecture for Containerized .NET Applications*. Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

[12]. Debski, A., Szczepanik, B., Malawski, M., Spahr, S., & Muthig, D. (2018). *A Scalable, Reactive Architecture for Cloud Applications*. *IEEE Software*, *35*(2), 62–71. https://doi.org/10.1109/ms.2017.265095722

[13]. Erl, T. (2007). *SOA Principles of Service Design (The Prenice Hall Service-Oriented Computing Series)*. In *Prentice Hall PTR eBooks*. Prentice Hall PTR. https://dl.acm.org/citation.cfm?id=1296147

[14]. Evans, E. (2014). *Domain-Driven Design Reference: Definitions and Pattern Summaries*. Dog Ear Publishing.

[15]. Evans, E. (2004). *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional.

[16]. Fields, J., Harvie, S., Fowler, M., & Beck, K. (2009). *Refactoring: Ruby Edition*. Pearson Education.

[17]. Fowler, M. (2010). *Domain-Specific Languages*. Pearson Education.

[18]. Fowler, M. (2012). *Fowler: Pattern Enterpr Applica Arch*. Addison-Wesley.

[19]. Garverick, J., & McIver, O. D. (2023). *Implementing Event-Driven Microservices Architecture in .NET 7: Develop event-based distributed apps that can scale with ever-changing business demands using C# 11 and .NET 7*. Packt Publishing Ltd.

[20]. Hippchen, B., Giessler, P., Steinegger, R. H., Schneider, M., & Abeck, S. (2017). *Designing Microservice-Based Applications by Using a Domain-Driven Design Approach*. International Journal on Advances in Software, 10, 432–445. https://www.thinkmind.org/articles/soft_v10_n34_2017_22.pdf

[21]. Huang, D., Xing, T., & Wu, H. (2013). *Mobile cloud computing service models: a user-centric approach*. *IEEE Network*, *27*(5), 6–11. https://doi.org/10.1109/mnet.2013.6616109

[22]. Indrasiri, K., & Suhothayan, S. (2021). *Design Patterns for Cloud Native Applications*. "O'Reilly Media, Inc."

[23]. Khononov, V. (2021). *Learning Domain-Driven Design*. "O'Reilly Media, Inc."

[24]. Kumar, V., & Agnihotri, K. (2021). *Serverless Computing Using Azure Functions: Build, Deploy, Automate, and Secure Serverless Application Development with Azure Functions (English Edition)*. BPB Publications.

[25]. Landre, E., Wesenberg, H., & Olmheim, J. (2007). Agile enterprise software development using domain-driven design and test first. *Conference on Object-Oriented Programming Systems, Languages, and Applications*. https://doi.org/10.1145/1297846.1297967

[26]. Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education.

[27]. Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.

[28]. Meyer, B. (1997). *Object-oriented Software Construction*. Prentice Hall.

[29]. Millett, S., & Tune, N. (2015). *Patterns, Principles, and Practices of Domain-Driven Design*. John Wiley & Sons.

[30]. Myers, B. (2022, January 5). *Red, Green, Refactor. What is Test-Driven Development*. Retrieved from:. https://medium.com/codecastpublication/red-green-refactor-what-is-test-driven-development-302794e06c [accessed: 29 September 2022].

[31]. Nguyen, P., Song, H., Chauvel, F., Muller, R., Boyar, S., & Levin, E. (2019). *Using microservices for non-intrusive customization of multi-tenant SaaS. In Foundations of Software Engineering*. https://doi.org/10.1145/3338906.3340452

[32]. Nilsson, J. (2006b). *Applying Domain-Driven Design and Patterns: With Examples in C# and .NET*. Pearson Education.

[33]. Palermo, J. (2019). *.NET DevOps for Azure: A Developer's Guide to DevOps Architecture the Right Way*. Apress.

[34]. Rademacher, F., Sachweh, S., & Zündorf, A. (2017). *Towards a UML Profile for Domain-Driven Design of Microservice Architectures*. In *Lecture Notes in Computer Science* (pp. 230–245). Springer Science+Business Media. https://doi.org/10.1007/978-3-319-74781-1_17

[35]. Rademacher, F., Sorgalla, J., & Sachweh, S. (2018). *Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective*. *IEEE Software*, *35*(3), 36–43. https://doi.org/10.1109/ms.2018.2141028

[36]. Steinegger, R. H., Giessler, P., Hippchen, B., & Abeck, S. (2017). *Overview of a Domain-Driven Design Approach to Build Microservice-Based Applications*. Conference: The Third International Conference on Advances and Trends in Software Engineering (SOFTENG 2017), 79–87. https://www.thinkmind.org/articles/softeng_2017_4_30_64138.pdf

[37]. Stuckenberg, S., Kude, T., & Heinzl, A. (2014). *Understanding the role of organizational integration in developing and operating Software-as-a-Service*. *Journal of Business Economics*, *84*(8), 1019–1050. https://doi.org/10.1007/s11573-013-0701-5

[38]. Uludağ, Ö., Hauder, M., Kleehaus, M., Schimpfle, C., & Matthes, F. (2018). *Supporting Large-Scale Agile Development with Domain-Driven Design*. *Lecture Notes in Business Information Processing*, 232–247. https://doi.org/10.1007/978-3-319-91602-6_16

[39]. Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley.

[40]. Vernon, V. (2016). *Domain-Driven Design Distilled*. Addison-Wesley Professional.

[41]. Vettor, R., Smith, S. (2023). *Architecting Cloud Native .NET Applications for Azure*. Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/

[42]. Villaça, L. A., Azevedo, L. G., & Baião, F. A. (2018). *Query strategies on polyglot persistence in microservices. In ACM Symposium on Applied Computing*. https://doi.org/10.1145/3167132.3167316

[43]. Wlaschin, S. (2018). *Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#*. Pragmatic Bookshelf.

[44]. Young, G. (2011). *Event Centric: Finding Simplicity in Complex Systems*. Addison-Wesley Professional.

[45]. Zimarev, A. (2019). *Hands-On Domain-Driven Design with .NET Core: Tackling complexity in the heart of software by putting DDD principles into practice*. Packt Publishing Ltd.