# Applying the domain driven design in cloud native computing

## Patterns –

DDD techniques that enable to find common patterns that can be reuse in cloud applications.

A more advanced CQRS system might also use Event-Sourcing (ES), which stores events in the domain model instead of the current-state data (César de la Torre, Bill Wagner, Mike Rousos).

An example of this kind of service is the ordering microservice from the eShopOnContainers reference application. This service implements a microservice based on a simplified CQRS approach. It uses a single data source or database, but two logical models plus DDD patterns for the transactional domain, as shown in Figure 7-2.

Diagram showing a high level Simplified CQRS and DDD microservice.



Figure 7-2. Simplified CQRS- and DDD-based microservice

The Logical "Ordering" Microservice includes its Ordering database, which can be, but doesn't have to be, the same Docker host. Having the database in the same Docker host is good for development, but not for production.

The application layer can be the Web API itself. The important design aspect here is that the microservice has split the queries and ViewModels (data models especially created for the client applications) from the commands, domain model, and transactions following the CQRS pattern. This approach keeps the queries independent from restrictions and constraints coming from DDD patterns that only make sense for transactions and updates, as explained in later sections.

The essence of those patterns, and the important point here, is that queries are idempotent: no matter how many times you query a system, the state of that system won't change. In other words, queries are side-effect free.

Therefore, you could use a different "reads" data model than the transactional logic "writes" domain model, even though the ordering microservices are using the same database. Hence, this is a simplified CQRS approach.

One such pattern is the Aggregate pattern, which we examine more in later sections. Briefly,

You might not always gain advantages from this pattern in queries;.

As shown in Figure 7-2 in the previous section, this guide suggests using DDD patterns only in the transactional/updates area of your microservice (that is, as triggered by commands). Queries can follow a simpler approach and should be separated from commands, following a CQRS approach.

in this case, something inside a microservice.

Different Bounded Contexts (BCs) will employ different patterns. They have different responsibilities, and that leads to different solutions. It is worth emphasizing that forcing the same pattern everywhere leads to failure. Do not use CQRS and DDD patterns everywhere. Many subsystems, BCs, or microservices are simpler and can be implemented more easily using simple CRUD services or using another approach.

There is only one application architecture: the architecture of the system or end-to-end application you are designing (for example, the microservices architecture). However,. Do not try to apply the same architectural patterns as CQRS or DDD everywhere.

## Design a DDD-oriented microservice

And that is explicit in the form of a microservice. The components within those boundaries end up being your microservices, although in some cases a BC or business microservices can be composed of several physical services. DDD is about boundaries and so are microservices.

First, you want to initially create the smallest possible microservices, although that should not be the main driver; you should

If two microservices need to collaborate a lot with each other, they should probably be the same microservice.

## Layers in DDD microservices

For example, an entity could be loaded from the database. Then, through a REST Web API, part of this information or a grouping of it with data from other entities can be sent to the client UI.

The point here is that the domain entity is contained within the domain model layer and should not be propagated to other areas that it does not belong to, like to the presentation layer.

Additionally, you need to have always-valid entities (see the [Designing validations in the domain model layer](https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-model-layer-validations) section) controlled by aggregate roots (root entities). Therefore, entities should not be bound to client views, because at the UI level some data might still not be validated. This reason is what the ViewModel is for. The ViewModel is a data model exclusively for presentation layer needs. The domain entities do not belong directly to the ViewModel. Instead, you need to translate between ViewModels and domain entities and vice versa.

When tackling complexity, , the aggregate root.

## The domain model layer

Eric Evans's excellent book [Domain Driven Design](https://domainlanguage.com/ddd/) says the following about the domain model layer and the application layer.

like Entity Framework or NHibernate. Ideally, your domain entities should not derive from or implement any type defined in any infrastructure framework.

Most modern ORM frameworks like Entity Framework Core allow this approach, so that your domain model classes are not coupled to the infrastructure. However, having POCO entities is not always possible when using certain NoSQL databases and frameworks, like Actors and Reliable Collections in Azure Service Fabric.

.

Also, this aspect does not mean you can take a model designed for a relational database and directly move it to a NoSQL or document-oriented database. In some entity models, the model might fit, but usually it does not. There are still constraints that your entity model must adhere to, based both on the storage technology and ORM technology.

### The application layer

Moving on to the application layer, we can again cite Eric Evans's book [Domain Driven Design](https://domainlanguage.com/ddd/):

A microservice's application layer in .NET is commonly coded as an ASP.NET Core Web API project. The project implements the microservice's interaction, remote network access, and the external Web APIs used from the UI or client apps. It includes queries if using a CQRS approach, commands accepted by the microservice, and even the event-driven communication between microservices (integration events). The ASP.NET Core Web API that represents the application layer must not contain business rules or domain knowledge (especially domain rules for transactions or updates); these should be owned by the domain model class library.

The application layer must only coordinate tasks and must not hold or define any domain state (domain model).

Basically, the application logic is where you implement all use cases that depend on a given front end. For example, the implementation related to a Web API service.

Most of all, the domain model layer must not directly depend on any infrastructure framework.

### The infrastructure layer

that use a DBContext to persist data in a relational database.

Your domain model layer class library should have only your domain code, just POCO entity classes implementing the heart of your software and completely decoupled from infrastructure technologies.

Thus, your layers or class libraries and projects should ultimately depend on your domain model layer (library), not vice versa, as shown in Figure 7-7.

As noted earlier, you can implement the most complex microservices following DDD patterns, while implementing simpler data-driven microservices (simple CRUD in a single layer) in a simpler way.



## In practice –

Every software project has a set of attributes, the most important of which are the amounts of data it operates on, performance requirements, business logic complexity, and technical complexity.

The techniques DDD proposes are useful for projects that have a lot of complex business rules. DDD won't help if a project needs to achieve outstanding performance or program against hardware systems. The only purpose DDD concepts serve is to tackle business logic complexity. A typical example of software with complicated business logic is enterprise-level applications. Most enterprise projects don't have outstanding performance requirements. Developers working on them usually don't have to deal with technical complexity on their own because there are plenty of tools that abstract out this kind of complexity for them. The most difficult challenge in such projects is dealing with the complexity of business logic in such a way that the solution can be extended and maintained in the long run. That is precisely the task that domain-driven design practices are intended to address. They help with creating code, which not only fully powers the problem but also does it in the simplest and thus most maintainable way possible.

... The beauty of domain-driven design is that its practices complement these two software development principles. It allows for the extraction of the central part of the problem domain and its simplification, removing most of the necessary complexity. The ability to express business logic in the clearest way possible is a single trait that makes domain-driven design so appealing in enterprise-level applications.

It is hard to estimate how important that is.

Keeping that complexity under control is the most difficult task in modern business line software. There is only so much complexity we can deal with at once. If the code exceeds it, it becomes really hard, and at some point even impossible, to change anything in the software without introducing some unexpected side effects. Extending such a project becomes a pain and usually results in a lot of bugs. This, in turn, slows down the development and may eventually lead to the failure of the project. One of the most common reasons for software project failure is uncontrollable complexity growth. Domain-driven design helps prevent it.

## Main Concepts of Domain-Driven Design

The first important concept is the notion of ubiquitous language, that is, the language structured around the domain model and used by all team members to refer to the elements of that domain. In many projects, domain experts and developers use different sets of terms when they talk about the domain. This difference leads to misunderstandings and slows down the overall development process. The notion of ubiquitous language helps eliminate the barrier.

Domain-driven design proposes that.

Domain-driven design is not only about writing code. Adhering to DDD practices also implies a heavy communication process between developers and domain experts. It's important to have direct access to the experts in the problem domain, because that's the only way to get complete information about the problem you are solving. To get the most out of domain-driven design, the domain knowledge should constantly be refined with the experts in the company; it shouldn't be a one-way process.

Software developers are often enthusiastic about coding tasks that regard building an infrastructure for a future project. Such tasks are often appealing because they present a technical challenge. Also, because of their technical nature, the knowledge acquired when solving them can be reused in other projects. All these make such activities compelling to many developers. But having in-depth domain knowledge will enable a programmer to do the best job possible. This knowledge will guide through the code. It will help to look at it from the domain expert's point of view. This skill is indispensable, as it allows you to combine the best of the two worlds: write technically correct code on the one hand and express domain knowledge with it on the other. Although problem domains are different from project to project, the skill of systematizing them with code is reusable.

## Onion Architecture and Domain Isolation

DDD notions form a construction named "onion architecture." It is so named because it resembles an onion with multiple layers and a core inside. The upper layers depend on the lower ones, but the lower layers don't know about the upper. It might seem similar to a classic onion-layer architecture with entities. The difference here is that onion architecture emphasizes the fact that the core part of this structure cannot depend on anything else except itself. It means that the core elements of our domain model should act in isolation from each other. 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, and finally, UI is the outermost layer, if, of course, the application contains a user interface. All work with a database should be encapsulated in repositories. They can refer to it either directly or through an ORM, but the general rule should remain. 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, and then they can contain a value object, or a value object can keep a reference to an aggregate root, but they 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 of the four core elements of the domain model is the separation of concerns. Entities, value objects, domain events, and aggregates carry the most important part of the application, its business logic. They don't contain all of it, of course. Repositories and factories can keep some of the business logic as well, but these four elements include most of it. In the situation where you have some elements so deeply involved in the problem domain representation, it is vital to keep them as free as possible from other duties. 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. 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.



building blocks of domain-driven design

## Modeling Best Practices

DDD principles dictate that the main model becomes the heart of the software; this fact entails a guideline for how to work with the application code base. In practice, it means that the development process should always start with modeling the core domain, even if there is no UI or database structure yet. The user interface and the database are important elements of the system as well, but the core domain is the most important part. The investments in the domain model pay off greatly over time. It also means that in a typical enterprise application, an infrastructure code is less important than the core domain. Make certain that the business logic remains intact and does not dissolve in the infrastructure code.

## Domain-Driven Design and Unit Testing

When it comes to unit testing, it's important to keep a balance between the amount of coverage and the amount of effort. 100% coverage is an expensive mark to reach, and it doesn't necessarily provide proportional value to the quality of your software. In most enterprise-level applications, the value distribution corresponds to the number of unit tests in this way. At some point, the value gained from the additional tests doesn't justify the resources invested. In practice, unit tests should cover only those parts of the code base that are the most significant to the application, and this is the innermost layer in the onion architecture: entities, value objects, aggregates, and domain events, the elements that contain most of the domain knowledge of your application. It's a good idea to get 100% or close to 100% as coverage of them. That is another reason why the core layer of the domain model should remain isolated from other parts of the application, such as the database, email service, and so on. A good separation of concerns helps create testable code, which doesn't require any mocks or other test doubles to be tested. If the unit tests are hidden in the database or some other external dependency, it's a strong sign the domain model is not properly isolated.

Integration testing should cover the other parts of the code base. They are automated tests which cover several pieces of the application at once. Implementing them should be an easy task due to the great isolation achieved for the domain model.

Summary

This article explains how DDD principles apply to specific types of cloud services as well as provides an overview of software design principles such as YAGNI, which stands for implementing only the functionality you need right now, and KISS, which proposes the use of the simplest solution possible. These principles can help greatly when going along with a project. The beauty of domain-driven design is that its principles perfectly align with breaking a problem into consumable chunks and reducing its complexity to a level where it's no longer hard to understand and implement. The article emphasizes the importance of communication with domain experts, which needs to be two-sided, and the importance of the domain knowledge itself. As well as the notion of onion architecture, unit testing and why the domain model should remain isolated.

## Overview of a Domain-Driven Design Approach to Build Microservice-Based Applications

Steinegger, Roland & Giessler, Pascal & Hippchen, Benjamin & Abeck, Sebastian. (2017). Overview of a Domain-Driven Design Approach to Build Microservice-Based Applications.

Roland H. Steinegger, Pascal Giessler, Benjamin Hippchen and Sebastian Abeck

* April 2017
* Conference: The Third International Conference on Advances and Trends in Software Engineering (SOFTENG 2017)
* At: Venice, Italy

The traditional approach, as discussed by Erl , suggests a technical and functional separation of services. In contrast, according to Evans [4], domain-driven design (DDD) provides the key concepts required to compartmentalize microservices

The DDD approach provides a means of representing the real world in the architecture,

/e.g., by using bounded contexts

representing organizational units [5], and also identifies and

focuses on the core domain; both of these characteristics lead

to improved software architecture qualityЛ

In microservice architectures, these bounded contexts are used to arrange and identify microservices (S. Newman, 2015). Using DDD is a key success factor in building microservice-based applications [].

Domain-driven design emphasizes that the application is necessary to determine the underlying domain logic of microservices ()

This article is structured as follows: In Section II, DDD and microservice architecture, including a general introduction to software architecture and development and other related concepts, are introduced. Section III classifies DDD and microservices and introduces the software development activities required in building microservice-based applications according to the requirements of DDD. In the next section, a case

study demonstrates the application of these activities within a software development process, including artifacts. The limitations discovered while applying the activities are described in Section V. A conclusion regarding the activities and possible future areas of inquiry is presented in Section VI.

II. FOUNDATION AND RELATED WORK

Domain-driven design is an approach that is used in application development where the domain model is the central artifact. Eric Evans introduced this approach in the book Domain-Driven Design and identified the essential principles.

### III. PROCESS

This section classifies the activities involved in of DDD and concepts related to microservice architectures; furthermore, the software development activities involved in building microservice-based applications using DDD are introduced. The activities discussed can be applied to various software process models. However, DDD requires one to continuously question and adapt one’s understanding of the domain.

Evans suggests a four layered architecture, consisting of the user interface, application, domain and infrastructure layers.

During the requirements elicitation and analysis, two subactivities take place: first, the information model, as part of the domain model, is created by “crunching knowledge” with domain experts; second, a prototype is designed and is discussed with both the user and customer. As both activities are closely related (when discussing prototypes, the knowledge of the domain gets deeper, and when discovering the information model, terms or workflows might change), we combined them

into a single activity.



Figure 3. Overview of the activities used in building microservice-based

Applications

### Requirements Elicitation and Analysis

The first activity is about understanding the needs of the user. Two non-chronological ordered activities take place in this phase: exploration of the domain and designing a prototype. These activities highly influence each other, e.g., the terms from the domain model are used in the prototype while new insights might change them

VI. CONCLUSION AND FUTURE WORK

DDD offers key concepts and activities to build applications based on a microservice architecture,

?? In a case study, we showed the application of the activities

in an agile software development process to build a thesis

management applications as part of the SmartCampus and gave

examples of the resulting artifacts. ??

DDD is about focusing on the domain including its concepts, their relationships and business logic. Microservice architecture is about arranging and dividing distributed software building blocks.We showed missing requirement specifications and missing artifacts with our classification.

A major advantage of DDD and microservices is the reuse of existing functionality.

In addition to this research topic, we will continue to focus on how we can systematically derive web APIs for microservices with quality aspects in mind such as evolvability. The web API also plays a significant role in discovering and reusing microservices in the context of a microservice landscape.

## Domain-driven design patterns: A metadata-based approach

Le, Duc & Dang, Duc-Hanh & Nguyen, Viet Ha. (2016). Domain-driven design patterns: A metadata-based approach. 247-252. 10.1109/RIVF.2016.7800302.

* November 2016
* Conference: 2016 IEEE RIVF International Conference on Computing & Communication Technologies, Research, Innovation, and Vision for the Future (RIVF)

The overall goal of domain-driven design (DDD) (E. Evans) is to design software (iteratively) around realistic domain model(s), which both thoroughly capture the domain requirements and are technically feasible for implementation. A core principle of object-oriented DDD [1] for achieving this goal is to

use a ubiquitous language that is structured based on the domain model(s).

in a target object-oriented programming language of choice

The high-level descriptions of the patterns given in [1], [2] make use of a number of generic object-oriented design patterns ([3]).

A domain-driven design pattern (DDDP) is a design pattern that addresses a domain modelling problem, is described in a structured format, and whose form is a template model that is expressed in a well-defined modelling language.

VII. CONCLUSION

In this paper, we presented a proposal for domain-driven design patterns that realise the high-level patterns of the DDD method. We argued why DDDPs are necessary and discussed the case for four core DDDPs that resolve four non-trivial, frequently-recurring design problems. We proposed to use a domain class modelling language (DCML) for expressing the DDDPs’ form. This language is based on UML and uses metaattributes to define design metadata. It has a number of key features that are suitable for expressing not only the domain model of DDD in general but the DDDP’s form. We extended DCML with new meta-attributes to support the DDDPs and described each core DDDP in detail. We showed, using an extended tool architecture of a previous work, how domainspecific

examples of the DDDPs are translated to physical class models for automatically generating software prototypes. We have used the core DDDPs in developing several practical software (one of which was used as the motivating example in this paper). Our plan for future work includes improving

the aforementioned tool to incorporate the support for the definition of DCML, domain models, and DDDPs.

## Towards a UML Profile for Domain-driven Design of Microservice Architectures

Rademacher, F., Sachweh, S., & Zündorf, A. (2017). Towards a UML Profile for Domain-Driven Design of Microservice Architectures. SEFM Workshops.

Among others, DDD provides modeling means for decomposing a domain into Bounded Contexts and expressing the relationships between them. With the recent emergence of Microservice Architecture (MSA), DDD again gains broad attention because a Bounded Context

naturally maps to a Microservice, which enables the application of DDD for MSA design.

(iv)

As a set of model-driven practices, techniques and principles for software design, DDD has been defined by Evans in 2004 [3]. With Microservice Architecture (MSA) as an architectural style for distributed, service-based software systems [Newman, S, 2015], that is gaining broad attention of both practitioners and scientists as of 2014 [Pahl, C., Jamshidi, P., 2016], the relevance of DDD recently increases. This is due to DDD providing various modeling patterns and techniques for the identification of coherent domain concepts and their encapsulation within conceptual boundaries

that might serve as foundation for MSA-based service decomposition [Newman, S, 2015].

In this section we elaborate on DDD as an approach to abstracting a domain in the form of structural domain models that describe structure and relationships of domain concepts [3]. We also describe the Bounded Context pattern that is commonly proposed for modeling services in MSA [Newman, S, 2015].

In DDD, a domain model is a rigorously organized, selective abstraction of conceptual knowledge about a domain or relevant parts of it [3]. notation to express domain models is not bound to a certain modeling language.

An important aspect of mapping a Bounded Context and its encapsulated domain objects to a Microservice implementation is the determination of the service interfaces on the basis of context relationships.

While the described mappings of Bounded Context relationships to Microservice interfaces are intuitive, several questions arise when taking the potential informality of structural domain models in DDD into account. First, besides Bounded Context relationships in the form of Associations between fragmented, probably shared domain objects, none of the surveyed domain models comprises constructs that specify technical characteristics of context interfaces for subsequent

service implementation (cf. Section 3). Among these are the assignment of protocols and message formats to prospective interface operations, as well as an approach for stating the type of action performed by an operation, e.g. read or update.

We therefore focus on transforming Bounded Contexts into services with regard to deriving service interfaces from associations between domain objects of different contexts.

## Designing Microservice-Based Applications by Using a Domain-Driven Design Approach

Hippchen, Benjamin & Giessler, Pascal & Steinegger, Roland & Schneider, Michael & Abeck, Sebastian. (2017). Designing Microservice-Based Applications by Using a Domain-Driven Design Approach. International Journal on Advances in Software (1942-2628). 10. 432 - 445.

The current trend of building web applications using microservice architectures is based on the domain-driven design concept. Among practitioners, domain-driven design is a widely accepted approach to building applications. Applying and extending the concepts and tasks of domain-driven design is challenging because it lacks a software development process description and classification within existing software development process approaches.

In microservice architectures, these bounded contexts are used to arrange and identify microservices [S. Newman, 2015]. Using DDD is a critical success factor in building microservice-based applications [S. Newman, 2015].

When applying DDD to the development of microservice based applications, several problems may arise, depending on the level of experience of the development team

However, it neither provides a detailed and systematic development process for applying these principles and patterns nor does it classify them into the field of software engineering. Classifying the activities, introduced by DDD, into the activities of a software development process could improve the applicability.

Domain-driven design emphasizes that the application is necessary to determine the underlying domain logic of microservices;

DDD is an approach that is used in application development where the domain model is the central artifact. Software architects and developers use the domain model as main source for software design and development. Furthermore, DDD focuses on the business logic of the customer’s domain and neglects technical aspects of the application.

In Evans’ approach to DDD, the central principle is to align the desired application with the domain model. The domain model shapes the “ubiquitous language” that is used among the team members and functions as a tool used to achieve this goal.

C. Requirements Elicitation with Behavior-Driven Development

Vogel et al. provide a comprehensive framework for the area of software architecture [18],

/ O. Vogel, I. Arnold, A. Chughtai, and T. Kehrer, Software Architecture:

A Comprehensive Framework and Guide for Practitioners. Springer

Berlin Heidelberg, 2011, URL: http://dx.doi.org/10.1007/978-3-642-

19736-9 [retrieved: 2017.11.30]./

which is used to classify microservices and DDD. Their architecture framework has six dimensions: 1) architectures and architecture disciplines, 2) architecture perspectives, 3) architecture requirements, 4) architecture means, 5) organizations and individuals and 6) architecture methods. The essential terms used in describing an architecture are: systems, which consist of software and hardware building blocks; a software building block can be a functional, technical or platform building block. Building blocks can also consist of other building blocks and may require them. The authors also introduce the concept of architecture views; their definition is influenced by the IEEE [19].

I. A. W. Group et al., “IEEE Recommended Practice for Architectural Description,” IEEE Std, vol. 1471, 1998.

Architecture views are part of the documentation that describes the architecture. Each view is motivated by stakeholders’ concerns. These concerns specify the viewpoints on the architecture and, thus, specify the views.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Domain-Driven Design Fundamentals

DDD has proven to be a very effective approach for managing complex requirements

DDD, it continues to evolve.

Domain‑Driven Design is a huge topic.

That's one of the great things about our industry. The more you know, the more you realize how much more there is you don't know. ‑In this module, we'll focus on the value of Domain‑Driven Design. You'll learn what the term represents and what problems DDD can help you with in your software building process. ‑Not only will we share the benefits of DDD, but we will be sure to highlight some of the potential drawbacks.

## Understanding the Value of Domain-Driven Design

Domain-driven design focuses on the problems of the business domain that are attempted to be solved. It's a critical shift from decades of focusing on how to store the data and then letting that drive how the software is designed.

When Eric Evans wrote his book, his goal was to understand what was behind the successes he had achieved with large-scale, complex software projects and what the patterns were.

DDD provides a clean representation of the problem in code that can be readily understood and verified through tests.

With DDD, it's just as important to understand what the stakeholders want as it is to work with them as full partners on a project.

The ultimate goal isn't to write code or even to build software, but to solve problems. Stakeholders are not interested in building software but in being successful at their mission. Software is a more efficient way to accomplish this.

DDD itself is a complex topic.

software teams should work on being able to talk to business people in order to free up their creative modeling.

Another core theme in DDD is to focus on a single subdomain at a time.

Many applications just try to do too many things at once, and adding additional behaviors gets more and more difficult and expensive. With DDD, you'll divide and conquer. By separating the problem into separate subdomains, each problem can be tackled independently, making the problem much easier to solve. The term modeling is important in DDD and refers to how each subdomain is deciphered and designed. The principle of separation of concerns plays a critical role in identifying the subdomains. Many applications spread the domain logic between the persistence layer and the user interface, making it much more difficult to test and to keep all of the business logic consistent.

## Exploring the Benefits and Potential Drawbacks of DDD

DDD helps the process focus on small, almost-independent pieces of the domain, so the software that comes out of it is more flexible. Small parts can easily be moved or modified with little or no side effects. The resulting software also tends to be more closely mapped to the stakeholder's understanding of the problem. DDD gives you a clear and manageable path through a very complex problem. Eric Evans says that DDD isn't good for problems with a lot of technical complexity but not much complexity in the business domain.

But all this comes at a cost. A lot of time could be spent talking about the domain and the problems that need to be solved, sorting out what is truly domain logic and what is just infrastructure. DDD could be overkill in some scenarios, such as for an application or subdomain that is solely a data-driven application, which doesn't need much more than a lot of CRUD logic.

## Inspecting a Mind Map of Domain-Driven Design



### A mind map diagram of how many of the concepts and patterns of DDD are interrelated

Evans refers to this as a navigation map, and it lays out all of the pieces of domain-driven design and how they relate to one another. This is a concept of the big picture. Many of these terms are described later on in the article.

Modeling the domain and its subdomains is an intense examination of the problem space. Another important part of modeling is figuring out which subdomain each of the "bounded contexts" belongs to. Another important aspect of modeling is identifying what's called "bounded contexts." Entities, value objects, aggregates, domain events, repositories, and more, and how they interact with each other, are supposed to be the result of modeling. Another important concept, driven by the need for clear, concise communication, is "ubiquitous language." There are terms that are commonly used when discussing a particular subdomain. They come from the problem space, not the software world, but everyone on the team needs to agree on them so that there isn't any confusion or misunderstanding caused by the different terms used by different people.

## Conversation with Eric Evans on Subdomains and Bounded Contexts

subdomains and bounded contexts are different. A subdomain shows how the business or domain activity is broken up, while a bound context shows how the software and its development have been organized.  Quite often, these will match up perfectly, but not always.

## Understanding Entities

Even though a DDD app is driven by behavior, objects are still needed. 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. Entities naturally accumulate a ton of functionality because they play such a crucial role in the system. 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 values, entities are things that were defined by a thread of continuity and identity.

## Value Objects

A value object has very specific characteristics. 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 concept has to be modeled as an entity, the entity should be designed so that it is more likely to hold values than to hold other entities. Value objects are a really good place to put methods and logic because reasoning can be done without side effects, especially with the complications that identity brings along—all those things that make logic tricky.

## Aggregates

Eric Evans writes in his book Domain-Driven Design that an aggregate is a group of related objects that are changed as a single unit. Aggregates consist of one or more entities and value objects that change together. Aggregates are treated as a unit for data changes. The entire aggregate's consistency needs to be considered before any changes are applied. 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. It's also up to the aggregate root to keep its invariants, like the number and type of its parts, the same. An invariant is a condition that should always be true for the system to be in a consistent state.

## Repository

A repository is a set of all the objects of a certain type that can be queried like a collection, but with more options. Repositories provide a common abstraction for all of our persistence concerns. This makes it easy for clients to get model objects and manage their lifecycle. The public interface of a repository very clearly communicates the design decisions. Only certain objects should be accessed directly, so repositories provide and control this access. An important benefit is that repositories make the code easier to test. They reduce tight coupling with external resources like databases, which would normally make unit testing difficult. When the code for accessing data is wrapped up in one or more well-known classes, it is easier and safer to use.

## Domain Events

Domain events are a critical part 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. In this way, the objects that are raising the events don't need to worry about the behavior that needs to occur when the event happens. And likewise, the event handling objects don't need to know where the event came from. Domain events are encapsulated as objects. 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 our ubiquitous language. Events are helpful because they signal that a certain thing has happened, so other parts of the system can „listen“ and take action accordingly. A domain event is essentially a message, a record of something that happened in the past.

## Clean Architecture: Patterns, Practices, and Principles

Using the modern patterns, practices, and principles listed above, a simple, understandable, adaptable, testable, and maintainable software architecture could be designed and built.

## Clean Code

"Clean architecture" is a philosophy of architectural essentialism.

Clean architecture is 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.

Clean architecture builds only what is necessary, when it is necessary.

Clean architecture optimizes for maintainability.

## Testable Architecture

Test-Driven Development is a software practice in which a failing test is created first before any production code is written and used to drive the design of the architecture. This three-step process of Test-Driven Development is referred to as "red, green, and refactor.":

red step - creating a failing test for the simplest piece of functionality

green step - Implementing enough production code to get that failing test to pass.

refactor step - improve both the test and production code to keep the quality high.

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.

A comprehensive collection of tests is produced by using test-driven development, which includes all of the application's code that is only crucial in a passive sense.

Additionally, by using TDD, the testing process is what guides the design of each of these classes and methods. Testable code is what produces maintainable code.

There are a variety of types of tests that exist in the world of software testing.

Some tests are based on the subject matter. Unit, integration, component service, and user interface tests, for instance. 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.

Mike Cohn talks about what he calls the "test automation pyramid" in his book Succeeding with Agile. 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 a large number of low-cost tests and a small number of high-cost tests.

## Modern Software Architecture: Domain Models, CQRS, and Event Sourcing

In the book Patterns of Enterprise Application Architecture, Martin displays a diagram that has complexity on the X axis and time and costs on the Y axis. The curve shows that at some point beyond a certain level of complexity following data-centric design patterns, even a small increase in complexity results in a significant peak in costs.



On the other hand, the time and cost of a project designed from a domain-centric perspective tended to grow linearly with complexity but had to deal with quite high startup costs.

In the end, DDD is exactly what its name implies: design that is based on the characteristics of the domain.

## Ubiquitous Language

It aims to build a common and business-oriented language. The primary goal of the language is to avoid misunderstandings and bad assumptions.

Using common terminology helps make better sense of user requirements.

It is thorough, smooth, and clear so that both domain experts and technical people can understand it.

Most of the time, the glossary is saved as an office document, which makes it an official part of the project documentation.

## Bounded Contexts

The bounded context is the limited space within the domain that gives each element of the ubiquitous language a clear, unique meaning.

fight ambiguity and duplication of concepts.

ideal tool to integrate legacy code and external components into the system.

## Event Sourcing

Event sourcing is a way of designing based on the idea that all changes to the state of an application over the course of its lifetime are stored as a series of events.  It ends up having serialized events as the building blocks of the application. The application's data source is essentially a stream of serialized events.

Most applications today work by storing the current state of domain entities and using that stored state as a starting point for business transactions. Rather than storing all of 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 containing a given list of items. This is actually an event-based representation of an entity.

An event is something that has happened in the past. Events are an expression of the ubiquitous language. Events could be kept in a NoSQL database, an ad-hoc relational table, or with a particular product. An event store of any kind is an append-only store and doesn't support deletions. Events express the entire state of a domain entity. To get the full state, the application timeline should be replayed from the beginning. This could sometimes involve handling excessive amounts of data. In this case, define snapshots, which are the state of the entity at a given time. Once stored, events are immutable. For scalability purposes, duplicate and replicate events are possible. With events, everything that occurred at the specific moment it occurred is recorded, regardless of the results it had. A lower abstraction level is used to store any system data.

Storage can be relational, document-based, graph-based.

The persistence of messages, which records all alterations to the application's state, is the key component of event sourcing.

The state of an aggregate can be recreated using recorded events.

Replay is just about looking into this data and performing logic to extract information from it.

Ad-hoc projections can address other, more interesting scenarios, like business intelligence, statistical analysis, what if, and why not simulation.

Yes, it might be hard and impractical to project state from logged events if there are a lot of them, and the number of logged events in many applications can only grow over time because it's an append-only store.

Instead of processing the entire stream of events, you serialize the state of aggregates at a given point in time and save that as a value. Next, you keep track of the snapshot point and replay events for an aggregate from the latest snapshot to the point of interest.

## Event-based Data Stores