DOMAIN-DRIVEN DESIGN AS A CHOICE FOR SOFTWARE DEVELOPMENT IN PROJECTS WITH COMPLICATED BUSINESS PROCESSES

Svetlana ANĐELIĆ[1], Goran RADIĆ[1], Nikola DRAGOVIĆ[1], Dušan UNKULOV[1]
[1]Information Technology School, Savski Nasip 7, New Belgrade, 11070, Serbia, Email: svetlana.andjelic@its.edu.rs, goran.radic@its.edu.rs, nikola.dragovic@its.edu.rs, dusan24314@its.edu.rs

Abstract

The article explains the critical concepts when using domain-led design as a software development approach. It presents what makes a domain-driven design, how that philosophy came about, the advantages and disadvantages, and when it is best to use it. All the necessary elements that are essential for software designed in this way to succeed, which together form a domain-driven design, are explained in detail. Particular emphasis is placed on the importance of domain-driven design in the case of software in projects with complicated business processes.

Keywords: *domain-driven design (DDD) software principles; onion architecture.*

JEL Classification: O33

Introduction

Software is a set of instructions that allow users to interact with a computer and perform specific tasks. [1] There are many such definitions, and this article will

not deal with which of these definitions is correct but will focus on making the software itself using specific methods. Software is just a means to an end, and usually, that goal is something convenient and accurate. For example, the software controls air traffic, which is directly related to the real world. Today, in the case of air traffic, sophisticated machines are used to manage the software to coordinate the flight of thousands of planes in the air at any given time.

However, one of the main problems for people making software of any nature is that they ignore software created to solve a real-life problem. So, the software must be practical and valuable. Otherwise, people would not invest so much time and resources in its creation. Therefore, it is highly connected with a particular aspect of human life. Thus, a specific software cannot be separated from the domain sphere for which it was built. This attitude leads to many problems, which this article will address.

Of course, there are different ways to approach software design. In the last 20 years, experts in the software industry have practised several methods for making their products, each with its advantages and disadvantages. The purpose of this article is to focus on a technique that has emerged and evolved over the last two decades but has crystallized more clearly over the last few years: "Domain-driven design" (DDD).

DDD is a development philosophy introduced initially by Eric Evans in Domain-Driven Design: Tackling Complexity in the Heart of Software (2003).

This article will explain why this approach to software development has become very popular in the industry in recent years. Each of the main ideological principles and practices of this philosophy will be covered.

1. Domain

It is essential to understand what a domain represents in the software world regarding domain-driven design. In the context of software engineering, a part usually refers to the subject area in which the software is intended to be used. In other words, during software development, the domain is the sphere of knowledge and activity on which the application's logic is based.

The example best shows the importance of knowing the domain for which the application is being created. If we look at the complete banking system that needs to be modelled in software, it is hard to imagine that this can be achieved without the help of people who understand this area of business. However, unfortunately, developers often focus on technical problems that arise when creating software, much less on issues significant to domain experts, bankers.

46

The banking system is best understood by the people inside, their experts, the employees who use it, and everyone who maintains a bank's ecosystem. They know all the details, tangible and problematic parts, and possible questions and rules. Therefore, it is always necessary to understand the domain for which the software is designed to provide the required value to experts in the field.

2. Domain-driven design determinants

Now that it is clear what role the domain plays, the next question is what is not domain-driven design. So DDD is the software development philosophy first proposed by Eric Evans in 2003, and it includes various strategic, philosophical, tactical and technical elements and is related to many specific practices. It is used to create complex, "enterprise" software, closely linking implementation with a particular domain's model and business concept processes. Therefore, its strategic value refers to the replication of business domain concepts into software artefacts. This achieves a natural separation of the whole code following business problems and avoids a prevalent mistake where you first think about how the software will be made and with what technologies. Only then try to adapt business concepts to the natural language (Native language) of selected tools.

If the representation of money in code is taken as an example, most solutions will use some decimal type according to the programming language. However, the trap lies because there will soon be a misunderstanding in the conversation between the developer and, say, a financial consultant who explains all the domain concepts necessary for the software to contain. For example, suppose a consultant is presented with a figure of decimal 50, which should represent money. In that case, there may be disagreement about what exactly that number means since people who deal with money must know which currency is in question and maybe on what day. So a simple float or decimal is nowhere near enough for situations like this. The solution to such problems will be explained in more depth in the "Values" section.

Having in mind the mentioned problems, Evans placed perhaps the most significant focus on developing the concept of the so-called ubiquitous language. The universal language forms the basis for successfully mapping a domain language into the software code itself. When we say "common", we mean using the same terms, phrases, and other domain concepts, in discussions with domain experts and in the code itself.

In summary, DDD is a philosophy of bringing software closer to the business domain, not the other way around. In addition to a clear definition of what DDD is, it is essential to emphasize what DDD is not: [2]

1. DDD is not tied to any technology, tool or programming language. Although there is much talk about how some languages are more suitable for using this development philosophy, at least some of the principles and methods presented by DDD can be applied in any technological setting.

2. It is not a software development methodology. DDD can be used just as well in an agile environment (Scrum, Kanban ...) as in the so-called "big design up front" models (waterfall, spiral, V-model ...)

3. Perhaps the most important thing to say is that DDD is not always the right choice. If it is necessary to make a simple application without complicated business processes, the introduction of DDD principles would significantly complicate the software project without the benefits that DDD otherwise provides.

3. Reasons to use DDD

When a new software system or model is developed, there will never be a "real" answer to all the questions that follow. When designing any system, it is necessary to determine whether it is justified to use a particular approach or technology or not. Also, this article will not provide accurate answers to all the questions that arise. The correct answer always depends on the context in which the software is developed and the domain for which it is produced.

One of the main problems when using DDD is deciding whether to use it at all. The rule is that using DDD will generally pay off if the domain for which the software needs to be designed is complex enough. Since it is pretty challenging to determine precisely where the boundary defines whether a part is complicated or not, Vaugh Vernon presented a table that can be used to conclude whether it is worthwhile to start with the DDD approach [3].

If the total number of all-rounded points is seven or greater, the use of a domain-guided design should be considered.

From the table, it can be concluded that DDD is most suitable when the domain for which the software is made is initially very complex. However, when it comes to Complexity, the most important thing is to understand that it is not essential if the domain is objectively complicated. However, people who develop software must understand it to such an extent that they can safely "insert" it into the software itself.

In addition to the initial Complexity, a prominent role is played by the number of changes and additions of new functionalities through software development over time, usually years. Suppose there is a tendency for software to have a long lifespan, which can mean different things for each project individually. In that case, it is

recommended to use DDD so that the system does not succumb to one of the many "anti-patterns". This would lead to the inability to add changes, correct errors, and generally to the dissatisfaction of employees assigned to maintain such a code.

Table 2-1. Reason for choosing a domain-driven design [3]

Is the project…	points
If the application is entirely data-oriented and qualifies for a pure CRUD solution, where each operation is a simple database query to create, read, update or delete, DDD is not required.	0
If the system only requires 30 or fewer business operations, it is probably pretty simple. However, this would mean that the application will not have more than 30 use cases, with each of these cases having only minimal business logic.	1
If the system has more than 30 use cases with more extensive business logic, it slowly enters the DDD territory.	2
Even if the application will not be complex in the beginning, will the Complexity grow over time? Unfortunately, this is often not known until the actual users start working with it.	3
It is expected that the application's functionality will change very often over the next few years, and it cannot be assumed how complicated these changes will be.	4
The domain is not understood by the people who need to create the software because it is new or complicated. This most likely means that it is complex or at least deserves in-depth analysis to establish the level of Complexity.	5
Total points:	**?**

However, there are situations in which the choice of whether to use DDD depends on circumstances that are not directly related to the scalability, sustainability, and general stability of the software throughout its life. One such example is the "lean" business development methodology, especially in "startup" cultures.

Lean startup is a business and product development methodology that shortens product development cycles and quickly discovers whether the proposed business model is sustainable. This is achieved by adopting a combination of business hypothesis-based experimentation, iterative product release, and validated learning. With this in mind, the question arises as to how DDD would then fit into this methodology, given the fact that it takes a lot of time and effort to apply the principles and practices that make DDD, and lean dictates that rapid experiments should provide feedback from by users who tested that quickly made prototype. In this case, it may be decided not to start the implementation using DDD immediately but to include it later if the prototype is successful. [4]

When the software should be ready is the next significant factor in deciding whether to engage in DDD. There are various reasons why software should be delivered "just then", and it greatly influences the decisions of engineers related to its development. The so-called "time to market" is one of the more common reasons. Specifically, when it comes to "lean", to test the market or target group with an impoverished software solution, and only when the vision of what should be the final product is concretized, resort to domain-led design.

Another less popular reason among engineers is that the company is directly influenced by the shareholders who profit the most or lose depending on how the application is placed on the market. As a result, investors often pressure teams to launch software sooner than software engineers would like. As a result, decisions are made to avoid applying a philosophy such as DDD at the start of development for fear of delays, and thus the consequences.

There is another good circumstance in which it is advisable not to use DDD initially, yet it includes shareholders and investors as the main reason. Namely, to get the affection and the necessary financial help in general, "startups" must show in action what they need financial resources for. Then it is much better to make a simple and unfinished prototype of the application or software itself, which shows the usefulness and value it brings. At the same time, it does not take months or even years to develop something that is still not certain to be used at all. Of course, if the so-called "proof of concept" passes, and it is determined that development needs to start, DDD again becomes a valid option.

4. Software complexity

In the previous section, a detailed explanation was given under which conditions DDD is offered as a perfect solution for a broad set of practices, ways of making and designing the components of a software solution. From this module onwards, the focus will be on each of these practices and how all of this is achieved. The problem mentioned in previous sections and that DDD best solves is the problem of increased Complexity throughout the system. An explanation of how this is performed will be provided below.

To be clear about what exactly Complexity means in this context, it is necessary to define it more deeply. Rich Hickey did a great job in "Simple Made Easy". He presented Complexity as nothing more than an intertwining of components or ideas in the same part of the system. Nevertheless, first, there is a crucial difference between the two types of Complexity in software engineering.

Fred Brooks, a 1986 Turing Award winner, explained this brilliantly in his "No Silver Bullet - Essence and Accident in Software Engineering". Brooks singled out two different types of Complexity: random Complexity and essential Complexity. [5]

1. Essential Complexity is the Complexity of the mentioned components in the software, but it is necessary to solve the problem itself. It arises because of a problem that needs to be solved, and nothing can remove it; if users want the program to do 30 different things, then those 30 things are necessary, and the program has to do those 30 other things. It forms the essence of the final software.

2. Random Complexity is, on the other hand, the intertwining of components and ideas, which are not necessary for solving problems. This Complexity is accidental and can occur for two reasons:

• because there has been an unnecessary importation of things, ignorance of how this can be avoided, or insufficient commitment to this problem.

• because an effort has been made to optimize further some part of the system using more advanced techniques. For example, optimizing batch data processing for faster processing or introducing a more complicated data compression algorithm.

As a result, the software is much harder to understand than expected and often leads to massive frustration among people who maintain such software. Therefore, it is essential to know when introducing additional Complexity of this type is necessary and avoid the standard error known as "premature optimization".

Figure 2-2 presents a simple graphical example that explains these two concepts.

Figure 2-2 shows the differences and the most common causes of these two types of Complexity. On the left side is the essential one, which creates the domain itself in the "real" world. It is intentional, necessary and crucial for the software to be valuable and successful.

On the other hand, there is an example of "legacy" code, mostly massive projects made over many years and even decades. The biggest drawback of such systems is that many engineers work on the software solution and often rotate. Rotation mainly happens because it is complicated for developers to understand what they need to do and make software changes, most prominent engineers. The latter has a vision of how the whole system works have left the company. This leads to a lot of error correction in the code, for which they are not even sure what it is doing at all. It is known that fixing mistakes is not particularly interesting for

developers, so they leave companies searching for something else that leads to the mentioned rotation. In this process, each programmer leaves his "trace", which often further complicates the part of the system he worked on and thus contributed to the random Complexity that everyone is trying to avoid. [7]
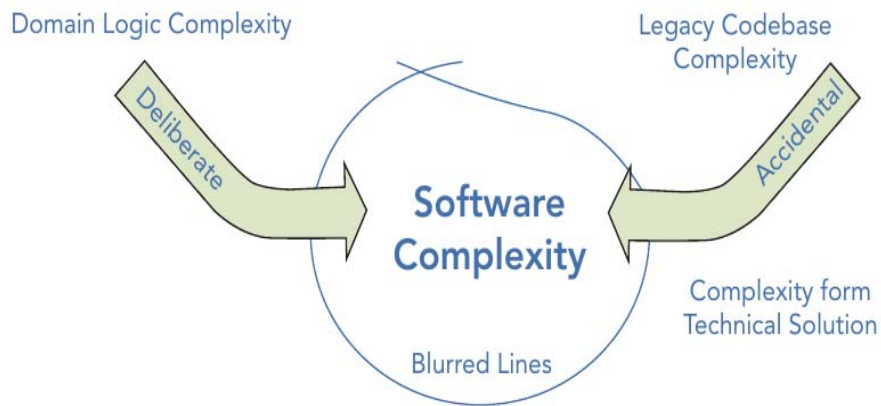

Figure 2-2. Software complexity [6]

5. Good and bad Complexity

Now that it is clear what Complexity is and how it is divided, the question arises of how to solve it. Perhaps the text from the previous section can be interpreted as "good" and "bad" Complexity, but that is not the case. Namely, if these two complexities were isolated, it would be established that each of them contributes to improving the software solution. If we discuss the first, introducing domain or so-called business logic into the answer, we get more enriched software with new functionalities.

An excellent example of this is the application for delivery of goods abroad. The client may request the introduction of a new domain logic that should adhere to the following:

"For each order, calculate an additional price that covers the cost of delivery to the final destination. This is achieved by automatically adding to the price before paying for the order, depending on the zone in which the address is located. "

This is what a request for new functionality or a so-called "user-story" might look like if it is a "scrum" methodology, which roughly defines what needs to be

done or added to an existing system. So, this is about the additional business logic necessary for the software to present how things work in the real world faithfully.

When it comes to random Complexity, it is much more often about the technical problems and solutions that make the software more efficient in various ways. They generally do not contribute to the business processes of the natural world but technically allow these processes to take place. The following client request describes this well:

"In the part of the internet portal for the delivery of goods, it is necessary to speed up the process of uploading photos of new products to reduce the load on the servers, and thus their maintenance cost."

At first glance, the simple requirement in terms of domain logic, moreover, has nothing to do with real-world problems, unlike in the past. However, this does not mean that it can be easily achieved. This requires some kind of data compression using one of the complex compression algorithms, which is necessary to keep the image quality at a certain level. This leads to the need to mainly include additional libraries in the code to solve such problems since it would be inefficient to write such things from the beginning. Therefore, introducing the component, which is very complex from the technological point of view, solves the problem but increases the mentioned random Complexity.

When both types of Complexity are viewed from this angle, both act as a necessary and desirable element of the software solution, as they solve the problems imposed by client requirements. However, the problem arises when mixing these complexities in the same part of the code. More precisely, when the infrastructural aspect of the code is combined with the domain code. An example of this is given in the next section.

6. Traditional "N-Layered" architecture

"N-Layered" architecture or "Multitier" is a concept of client-server architecture in software engineering. The functions of presentation, processing and data management are logically and physically separated into several layers. The rule is that the "above" layer cannot reference the "below" layer, which means "The component in the reference layer must not depend on the component from the reference layer and must function independently of any reference layer". The key to understanding here is that the "Data" layer must exist and function on its own, then the "Logical" layer exists and works on its own, but only by connecting to the base "Data" layer, and so on. An example can be seen in Figures 2-3.

Issue 1/2022

N Tier Architecture

Presentation Layers
- ASP.NET
- Silvelright
- WPF
- Console application

Business Layers
- WCF Service Layer
- EF ORM Layer
- Data Access Layer
- Business Logic Layer

Data Layers
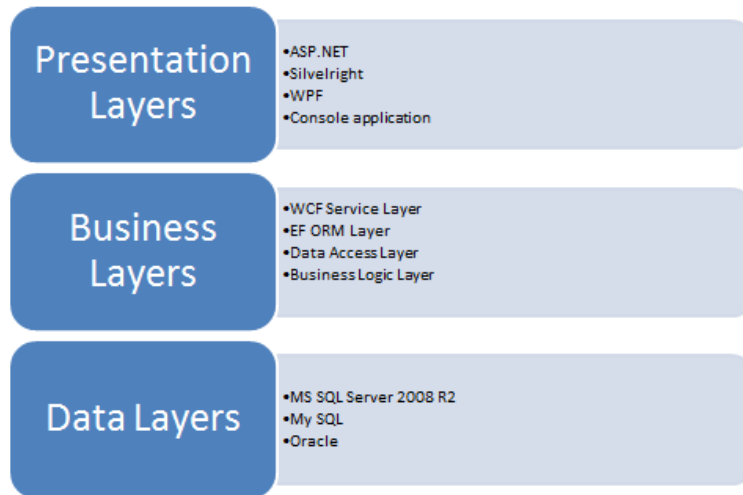- MS SQL Server 2008 R2
- My SQL
- Oracle

Figure 2-3. N-Tier architecture [8]

Layered architecture initially emerged to scale Internet-based client-server applications that can support hundreds of thousands of users. By placing the Load balancer level at the top of the processing logic, the system would withstand high loads more efficiently and provide a higher degree of resilience. That is how it worked, which is why it became so popular. [9]

However, over time, it has proven too inflexible to scale more modern effectively, low-latency applications where data volumes are exponentially more significant each year. The main reason why this is so may not be immediately apparent, as most of the flaws only come to the fore when the project grows. The reasons are as follows: [10]

1. Too generic solution

Division of the system into rigid layers tends to jeopardize flexibility. For example, a layered design may require validation in the middle layer when it is sometimes outstanding to apply the same validation logic in the presentation and intermediate layers in more straightforward cases. In addition, the reason closely

54

related to the data being processed is perhaps even closer to the data warehouse. The point is that the solution should satisfy specific business functionalities and not adhere to a predefined abstraction.

2. The problem of separation of worries

"Separation of Concerns (SoC)" is one of the most basic principles in software development. It is so important that 2 of the 5 SOLID principles are based on this principle:

- Single Responsibility
- Interface Segregation

Since the components in this architecture are generic, it is tough to define adequate abstractions. Layers tend to be restricted by their technical role rather than business functionality, which further leads to the "omission" of business logic in multiple places within a single layer, thus totally losing the authority of a particular business process by its "owner". After a while, it becomes impossible to tell where things are going wrong, and even banal changes require code changes on each layer. Adding new features becomes almost impossible, and most of the time is spent not crashing the rest of the system. Anti-patterns such as "Big ball of mud" (lack of clear architecture in the system) and "Shotgun Surgery" (changing one component means changing many others) are becoming a reality.

3. Tight Coupling

Another big problem is that in this architecture, the dependence of the components (eng. Dependency graph) always goes down. All of these classes and methods are tightly coupled due to the direct instantiation of layer objects below the current one. An example of this is using the new keyword in languages such as C # or Java. Because of this, every change in the class or method in the last layer causes the need to change and every component that directly depends on it. In small systems, it is easier to keep track of all these changes. However, as soon as the project grows, the vision of which components need to be changed is quickly lost. Thus, a "Shotgun Surgery" scenario is obtained (changing one feature imposes many varying others).

The opposite of this harmful practice is Loose Coupling, which achieves increased stability, sustainability and more remarkable ability to test a particular module. This is achieved by using the interface as a "contract" of communication between the components. Thus, the part implementing the interface can be changed without consequences as long as all these interfaces required is further complied with.

In the IT industry, as Uncle Bob, Robert Martin released a article entitled "The Dependency Inversion Principle" in the 1990s that explained in detail exactly how to solve all the problems mentioned above present using the "N-tier" architecture.

The Dependency Inversion Principle is the letter "D" in the "SOLID" principles, which is also credited to Uncle Bob and is still the best way to separate the dependencies of software components. The whole idea consists of two parts: [11]

1. High-level modules should not depend on low-level modules. Both should depend on abstraction.

2. Abstractions should not depend on details. Details should depend on abstraction.

An important detail of this definition is that high and low-level modules depend on abstraction. This design principle changes the direction of dependence and divides the support between high and low-level modules by introducing abstraction. In the end, two dependencies are obtained:

1. A high-level module depends on abstraction and
2. A low-level module depends on the same abstraction.

It is now clear that the "N-tier" architecture is not subject to respect for this design principle. It is easy to see that at the bottom of the chain of dependence is just a specific implementation or the already mentioned implementation detail database. This is precisely why alternative architectures were sought, to separate support from the database and respect the "Principle of Inversion Inversion" and other recommended already mentioned practices.

The previous section said that an example of an uncontrolled increase in Complexity in the software would be given by mixing the essential and the random. Modelling software that relies heavily on the database and its limitations is the case. The biggest problem is that the crucial part of the software, the domain, directly depends on the database's infrastructural detail. It has already been said that all the tools used in the software are used to enable the execution of a business process. However, in doing so, they add considerable random Complexity to the system. Databases are known to be very complex and certainly contribute the most to it. One example is storing "bool" values in a system. Although it is logical to use the native type in any programming language: true or false, in some databases such as "SQLServer", this is expressed with the bit data type, using 0 or 1. In such situations, developers often agree to compromise and use shorting or another alternative to solve this as painlessly as possible since the base is in the foreground. However, its limitations cannot be easily ignored. DDD provides a solution and uses the mentioned principles to put the database in the background.

7. Onion architecture

It has already been pointed out that DDD does not provide a ready-made solution that everyone must adhere to completely. However, Eric Evans put together and created this approach to software development, a set of best practices, design principles, and patterns in the IT industry.

One of these principles is the "Onion" architecture, first introduced by Jeffrey Palermo in 2008, currently providing the best alternative to the traditional "N-tier" approach. As he says, the "Onion" architecture is not suitable for small websites. However, it is ideal for long-term business applications as well as for applications with many complex processes. The architecture emphasizes the use of the interface, which achieves the already mentioned inversion of component dependencies and forces the externalization of the infrastructure, thus avoiding dependencies and reducing Complexity. Figure 2-4 shows what it looks like.
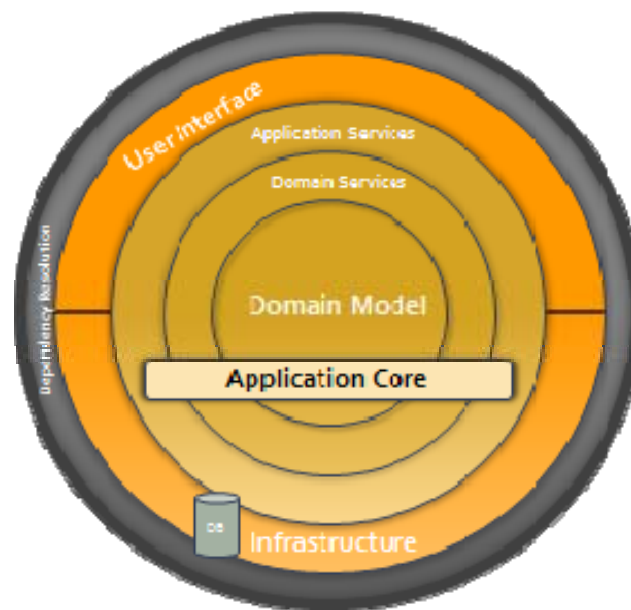

Figure 2-4. Onion architecture [12]

Issue 1/2022

The basic rule is that the code can depend on the more central layers, but it cannot rely on the further layers from the core. This architecture is fearlessly biased towards object-oriented programming and puts objects ahead of everything else. This is a bit reminiscent of the "N-tier" model, where the dependence also moves downwards. However, this time, there is a domain model in the centre. The domain model combines states and behaviours that model the organization for which the software is created. There are other layers with more behaviour around the domain model. The number of layers in the application core varies, but the most important thing is that the domain model is always in the centre. This means that the domain model is connected exclusively and only with itself. In other words, there is no unnecessary connection with things that change very often over time. For example, the "Angular framework" used to create the user interface releases new major versions every six months, often changing many things. Furthermore, having in mind that the software should live for years, and even decades, it is clear that it is essential to separate it as far as possible from the centre of the architecture and not use it as a support on which the software is built.

In this regard, Figure 2-4 clearly shows that the database is located outside, at the same level as the other infrastructure code. Database externalization can be quite a significant change compared to the "N-tier" approach where the database is in the foreground. Some applications can use a database as a data storage tool, but only through some external infrastructure code that implements an interface that makes sense for the application core. Separating an application from a database, file system, and physical computer reduces maintenance costs over the life of the entire application.

Of course, it is still necessary that the infrastructure communicates with the domain model so that the data is stored in the system according to what is required. That is why the "Onion" architecture is based on the principle of dependence inversion ("DiP"). The application core uses predefined interfaces, which maintains a loose dependency. Implementing these interfaces is on the very edge of "Onion", which includes complicated technical details, external libraries, databases, file systems, web protocols and everything else. This is accomplished by injecting that implementation code when the software code is executed, known as a runtime. [13]

There are other variations on this architecture, but they are also very similar with minor terminological differences. The most famous is Uncle Bob's "Clean Architecture" from 2003 and Alistair Cockburn's "Hexagonal architecture" from 2006. Using this architecture, DDD solves the main design problem that the

industry has encountered in the past. This is also the basis of all the elements that make up DDD, which will be discussed later in the next section.

8. DDD components

The last section presents an architecture that solves design problems that people have encountered in the past. The focus is on how to achieve a loose relationship between the components, the so-called "low coupling, high level of cohesion" ("Low coupling, high cohesion"), which according to many, is the crucial thing when it comes to creating a sustainable and scalable system. But, in the picture that shows the "Onion" architecture, there are other components that were not mentioned. These and many other elements play an essential role in what DDD wants to achieve.

It has been said that DDD is nothing but a set of practices, patterns and principles that together make a sustainable system. The creator of DDD, Eric Evans, recognized all these elements very well and presented them all in his book [14]

One of the reasons why Eric's book is not recommended for engineers who are just entering the realm of DDD is that it is tough to understand all the concepts covered in the book, having in mind a lot of new ideas. Figure 2-5 shows this. Although not trivial, everything becomes much clearer when each of these concepts is studied, and the value of their application in a software solution brings excellent results.

To make it easier to understand what all this represents, these practices and patterns can be divided into two groups: [15]

1. Strategic DDD:

has a broader context and is very influential in what kind of results the whole project will bring, mainly in the context of its scalability and the possibility of significant changes in the system. Strategic design is fundamental at the beginning of the project, where certain boundaries between the components are clearly defined, so it can be said that such decisions are not made every day.

2. Tactical DDD:

On the other hand, tactical DDD is something that developers encounter every day and contributes the most to the ability to adequately express a domain within the boundaries defined by strategic design. If all the principles are used in the right way, the possibility of making changes without fear of affecting another component is achieved. In addition, it provides a great way to test business logic and processes.

Conclusion

Looking at domain-driven design as a software development philosophy, it can be established that several complicated concepts must be understood for this philosophy to be successfully applied. The benefits of using DDD are evident if there is a need for it. In case the project contains complicated business processes that must be presented in the software, DDD is undoubtedly worth using. However, by its nature, it adds Complexity in itself, but it produces stable and scalable software in return. On the other hand, if it is necessary to create a more straightforward application, all DDD principles are unnecessary. Again, this does not mean that specific ideas from DDD cannot be used where necessary.

Probably the most critical part of the DDD philosophy is the proper use and understanding of strategic design. Creating a ubiquitous language together with domain experts is something without which the project would not be successful. This brings more readable code, clearer intent, and fewer comments explaining what each part of the system is doing. Finding the proper context boundaries and representing subdomains within these limited contexts is not an easy task, but it is a vital step towards software sustainability in the long run. In addition, this provides the possibility of total autonomy not only of the code but also between multiple teams of people who can work in other technologies, which significantly increases the options.

The final decision as to whether DDD should be used for a particular software is undoubtedly up to the one who designs it. In this article, many reasons are expressed as to whether such a decision is justified or not.

References

[1]   *TechTarget,   "Software   definition"   [Online].   Available: https://searchapparchitecture.techtarget.com/definition/software. [Accessed May 2020].*

[2] Airbrake," domain-driven design" [Online]. Available: https://airbrake.io/blog/software-design/domain-driven-design [Accessed May 2020]

[3] Vaughn Vernon, Implementing Domain-Driven Design, Feb, 2013, Chapter 1, Getting Started with DDD, p.1-42

[4] Planview, "Lean methodology" [Online]. Avaliable: https://www.planview.com/resources/articles/lean-methodology/ [Accessed May 2020].

[5]   WorryDream,   "no   silver   bullet"   [Online].   Available: http://worrydream.com/refs/Brooks-NoSilverBullet.pdf [Accessed May 2020].

[6] Medium, "Domain-Driven Design First impressions" [Online]. Available: https://labs.bawi.io/domain-driven-design-first-impressions-ca7275d90585 [Accessed May 2020]

[7] UnderstandLegacyCode, "What is Legacy Code? Is it code without tests?" [Online]. Available: https://understandlegacycode.com/blog/what-is-legacy-code-is-it-code-without-tests/ [Accessed May 2020].

[8] Bhrnjica, "WCF in N Tier Architecture" [Online]. Available: https://bhrnjica.net/2011/05/16/wcf-in-n-tier-architecture-techday-2011 / [Accessed May 2020]

[9] Stackify, "What is N-Tier Architecture?" [Online]. Available: https://stackify.com/n-tier-architecture/ [Accessed May 2020].

[10] Ben Morris, "The problem with tiered or layered architecture" [Online]. Available: https://www.ben-morris.com/the-problem-with-tiered-or-layered-architecture/ [Accessed May 2020].

[11] Stackify, "SOLID Design Principles Explained: Dependency Inversion Principle" [Online]. Available: https://stackify.com/dependency-inversion-principle/ [Accessed May 2020].

[12] StackExchange[Online]. Available: https://softwareengineering.stackexchange.com/questions/180499/presentation-vs-application-layer-in-ddd [Accessed May 2020]

[13] Jeffrey Palermo, "The Onion Architecture: part 1" [Online]. Avaliable: https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/ [Accessed May 2020].

[14] Domain-driven design, "Learning map" [Online]. Available: https://thedomaindrivendesign.io/learning-map/ [Accessed May 2020]

[15] Eric Evans, Tackling Complexity in the heart of the software, Aug 2003, Part II, Building Blocks of a Model-Driven Design, p.49-132.