PART I

The Principles and Practices of Domain-Driven Design

1

What Is Domain-Driven Design?

WHAT'S IN THIS CHAPTER?

- An introduction to the philosophy of Domain-Driven Design
- The challenges of writing software for complex problem domains
- How Domain-Driven Design manages complexity
- How Domain-Driven Design applies to both the problem and solution space
- The strategic and tactical patterns of Domain-Driven Design
- The practices and principles of Domain-Driven Design
- The misconceptions of Domain-Driven Design

Domain-Driven Design (DDD) is a development philosophy defined by Eric Evans in his seminal work *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley Professional, 2003). DDD is an approach to software development that enables teams to effectively manage the construction and maintenance of software for complex problem domains.

This chapter will give you a high-level introduction to DDD's practices, patterns, and principles along with an explanation of how it will improve your approach to software development. You will learn the value of analyzing a problem space and where to focus your efforts. You will understand why collaboration, communication, and context are so important for the design of maintainable software.

At the end of this chapter you will have a solid understanding of DDD that will provide context to the detail of the various patterns, practices, and principles that are contained throughout this book. However, before we delve into how DDD handles complexity it's important to understand what problems can cause software to get into an unmanageable state.

THE CHALLENGES OF CREATING SOFTWARE FOR COMPLEX PROBLEM DOMAINS

To understand how DDD can help with the design of software for a nontrivial domain, you must first understand the difficulties of creating and maintaining software. By far, the most popular software architectural design pattern for business applications is the Big Ball of Mud (BBoM) pattern. The definition of BBoM, as defined by Brian Foote and Joseph Yoder in the paper "Big Ball of Mud," is "… a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle."

Foote and Yoder use the term BBoM to describe an application that appears to have no distinguishable architecture (think big bowl of spaghetti versus dish of layered lasagna). The issue with allowing software to dissolve into a BBoM becomes apparent when routine changes in workflow and small feature enhancements become a challenge to implement due to the difficulties in reading and understanding the existing codebase. In his book, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley Professional, 2003), Eric Evans describes such systems as containing "code that does something useful, but without explaining how." One of the main reasons software becomes complex and difficult to manage is due to the mixing of domain complexities with technical complexities, as illustrated in Figure 1-1.

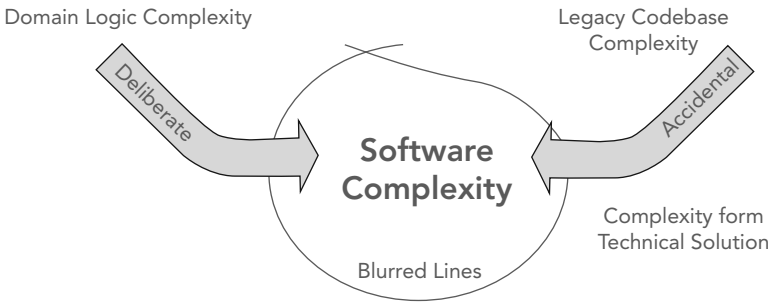

FIGURE 1-1: Complexity in software.

Code Created without a Common Language

A lack of focus on a shared language and knowledge of the problem domain results in a codebase that works but does not reveal the intent of the business. This makes codebases difficult to read and maintain because translations between the analysis model and the code model can be costly and error prone.

Code without a binding to an analysis model that the business understands will degrade over time and is therefore more likely to result in an architecture that resembles the BBoM pattern. Due to the cost of translation teams that do not utilize the rich vocabulary of the problem domain in code will decrease their chances of discovering new domain concepts when collaborating with business experts.

WHAT IS AN ANALYSIS MODEL?

An analysis model is used to describe the logical design and structure of a software application. It can be represented as sketches or by using modeling languages such as UML. It is the representation of software that non-technical people can conceptualize in order to understand how software is constructed.

A Lack of Organization

As highlighted in Figure 1-2, the initial incarnation of a system that resembles BBoM is fast to produce and often a well-rounded success, but because there is little focus based on the design of an application around a model of the problem domain, subsequent enhancements are troublesome. The codebase lacks the required synergy with the business behavior to make change manageable. Complexities of the problem domain are often mixed with the accidental complexities of the technical solution.

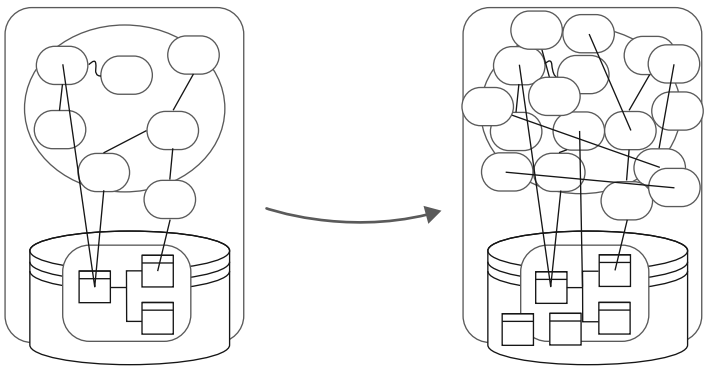

FIGURE 1-2: Code rot.

The Ball of Mud Pattern Stifles Development

Continuing to persist with an architectural spaghetti-like pattern can lead to a sluggish pace of feature enhancement. When newer versions of the product are released, they can be buggy due to the unintelligible mess of the codebase that developers have to deal with. Over time, the development team increasingly complains about the difficulty of working in such a mess. Even if resources are added to the project, velocity cannot be increased to a level that satisfies the business.

In the end, exasperated by the situation, the request for the dreaded application rewrite is granted. Without due care and consideration, however, even the greenfield project can fall foul of the same issues that created the original BBoM. This entire experience can be frustrating for the business that saw a great return on investment (ROI) in terms of features and speed of delivery at the beginning but over time, even with additional investment in resources, did not see the sustained evolution of the product to meet their needs. Ultimately the BBoM is bad news for you as a developer because it's a messy bug-prone code base that you hate dealing with. And it's bad news for the business because it reduces their capability to rapidly deliver business value

A Lack of Focus on the Problem Domain

Software projects fail when you don't understand the business domain you are working within well enough. Typing is not the bottleneck for delivering a product; coding is the easy part of development. Outside of non-functional requirements creating and keeping a useful software model of the domain that can fulfill business-use cases is the difficult part. However, the more you invest in understanding your business domain the better equipped you will be when you are trying to model it in software to solve its inherent business problems.

WHAT IS A PROBLEM DOMAIN?

A problem domain refers to the subject area for which you are building software. DDD stresses the need to focus on the domain above anything else when working on creating software for large-scale and complex business systems. Experts in the problem domain work with the development team to focus on the areas of the domain that are useful to be able to produce valuable software. For example, when writing software for the health industry to record patient treatment, it is not important to learn to become a doctor. What is important to understand is the terminology of the health industry, how different departments view patients and care, what information doctors gather, and what they do with it.

HOW THE PATTERNS OF DOMAIN-DRIVEN DESIGN MANAGE COMPLEXITY

DDD deals with both the challenge of understanding a problem domain and creating a maintainable solution that is useful to solve problems within it. It achieves this by utilizing a number of strategic and tactical patterns.

The Strategic Patterns of DDD

The strategic patterns of DDD distil the problem domain and shape the architecture of an application.

Distilling the Problem Domain to Reveal What Is Important

Not all of a large software product needs be perfectly designed—in fact trying to do so would be a waste of effort. Development teams and domain experts use analysis patterns and knowledge crunching to distill large problem domains into more manageable subdomains. This distillation reveals the core sub domain—the reason the software is being written. The core domain is the driving force behind the product under development; it is the fundamental reason it is being built. DDD emphasizes the need to focus effort and talent on the core subdomain(s) as this is the area that holds the most value and is key to the success of the application.

This clarity on where to focus effort can also empower teams to look for open source off-the-shelf solutions for some of the less important parts of a system, which means that they have more time to focus on what is important and ensure that the core domain does not become a BBoM.

Discovering the core domain helps teams understand why they're producing the software and what it means for the software to be successful to the business. It is the appreciation for the business intent that will enable the development team to identify and invest its time in the most important parts of the system. As the business evolves, so in turn must the software; it needs to be adaptable. Investment in code quality for the key areas of an application will help it change with the business. If key areas of the software are not in synergy with the business domain then, over time, it is likely that the design will rot and turn into a big ball of mud, resulting in hard-to-maintain software.

Creating a Model to Solve Domain Problems

In the solution space a software model is built for each subdomain to handle domain problems and to align the software with the business contours. This model is not a model of real life but more an abstraction built to satisfy the requirements of business use cases while still retaining the rules and logic of the business domain. The development team should focus as much energy and effort on the model and domain logic as it does on the pure technical aspects of the application. To avoid accidental technical complexity the model is kept isolated from infrastructure code.

All models are not created equal; the most appropriate design patterns are used based on the complexity needs of each subdomain rather than applying a blanket design to the whole system. Models for subdomains that are not core to the success of the product or that are not as complex need not be based on rich object-oriented designs, and can instead utilize more procedural or data-driven architectures.

Using a Shared Language to Enable Modeling Collaboration

Models are built through the collaboration of domain experts and the development team. Communication is achieved using an ever-evolving shared language known as the ubiquitous language (UL) to efficiently and effectively connect a software model to a conceptual analysis model. The software model is bound to the analysis model by using the same terms of the UL for its structure and class design. Insights, concepts, and terms that are discovered at a coding

level are replicated in the UL and therefore the analytical model. Likewise when the business reveals hidden concepts at the analysis model level this insight is fed back into the code model; this is the key that enables the domain experts and development teams to evolve the model in collaboration.

Isolate Models from Ambiguity and Corruption

Models sit within a bounded context, which defines the applicability of the model and ensures that its integrity is retained. Larger models can be split into smaller models and defined within separate bounded contexts where ambiguity in terminology exists or where multiple teams are a working in order to further reduce complexity.

Bounded contexts are used to form a protective boundary around models that helps to prevent software from evolving into a BBoM. This is achieved by allowing the different models of the overall solution to evolve within well-defined business contexts without having a negative, rippling impact on other parts of the system. Models are isolated from infrastructure code to avoid the accidental complexity of merging technical and business concepts. Bounded contexts also prevent the integrity of models being corrupt by isolating them from third-party code.

Compare the diagram in Figure 1-3 to Figure 1-2. The diagram shows how the strategic patterns of DDD have been applied to the software to manage the large problem domain and protect discrete models within it.

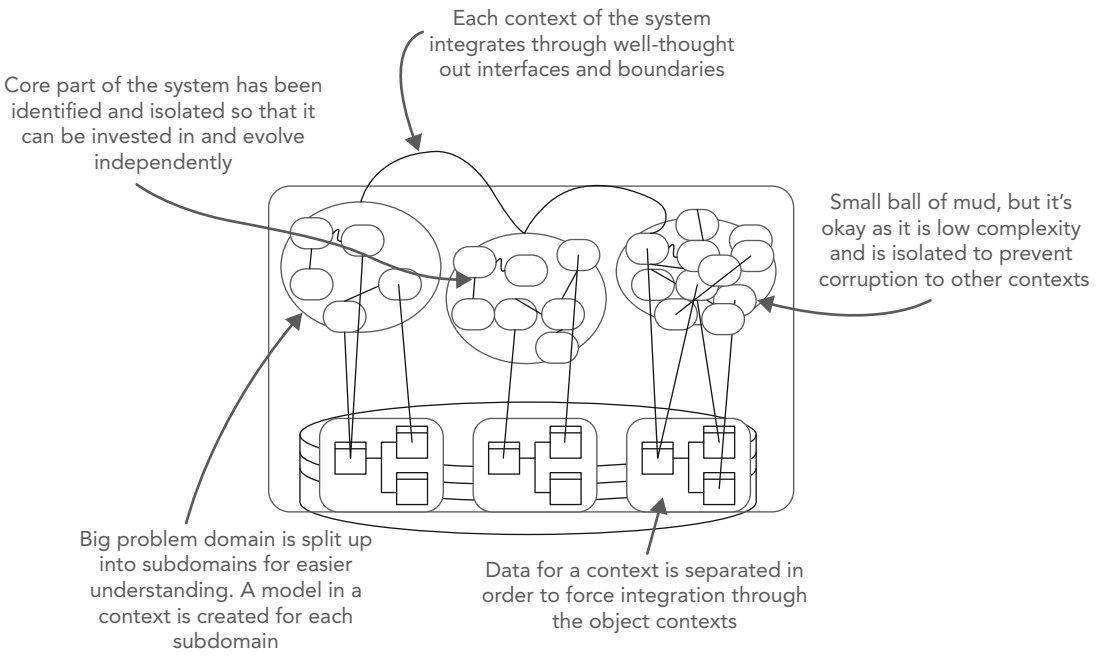

FIGURE 1-3: Applying the strategic patterns of Domain-Driven Design.

THE BIG BALL OF MUD IS NOT ALWAYS AN ANTIPATTERN

Not all parts of a large application will be designed perfectly—nor do they need to be. Although it's not advisable to build an entire enterprise software stack following the BBoM pattern, you can still utilize the pattern. Areas of low complexity or that are unlikely to be invested in can be built without the need for perfect code quality; working software is good enough. Sometimes feedback and first-to-market are core to the success of a product; in this instance, it can make business sense to get working software up as soon as possible, whatever the architecture. Code quality can always be improved after the business deems the product to be a success and worthy of prolonged investment. The key to reaping the benefits of the BBoM is to define a context around the bounded contexts that use the BBoM to avoid them corrupting the core subcomain.

Understanding the Relationships between Contexts

DDD understands the need to ensure that teams and the business are clear on how separate models and contexts work together in order to solve domain problems that span across subdomains. Context maps help you to understand the bigger picture; they enable teams to understand what models exist, what they are responsible for, and where their applicability boundaries are. These maps reveal how different models interact and what data they exchange to fulfill business processes. The relationships between the connections and more importantly the grey area of process that sits between them is often not captured or well understood by the business.

The Tactical Patterns of DDD

The tactical patterns of DDD, also known as model building blocks, are a collection of patterns that help to create effective models for complex bounded contexts. Many of the coding patterns presented within the collection of tactical patterns have been widely adopted before Evans's text and catalogued by the likes of Martin Fowler in *Patterns of Enterprise Application Architecture* and Erich Gamma, et al. in *Design Patterns: Elements of Reusable Object-Oriented Software*. These patterns are not applicable to all models, and each must be taken on its own merit with the correct architectural style applied.

The Problem Space and the Solution Space

All of the patterns detailed in this section help to manage the complexity of a problem—aka the problem space or they manage complexity in the solution—aka the solution space. The problem space, as shown in Figure 1-4, distils the problem domain into more manageable subdomains. DDD's impact in the problem space is to reveal what is important and where to focus effort. In the next chapter we will look in more detail on the patterns that can help reduce complexity in the problem space.

The solution side of DDD, shown in Figure 1-5, covers patterns that can shape the architecture of your applications and make it easier to manage.

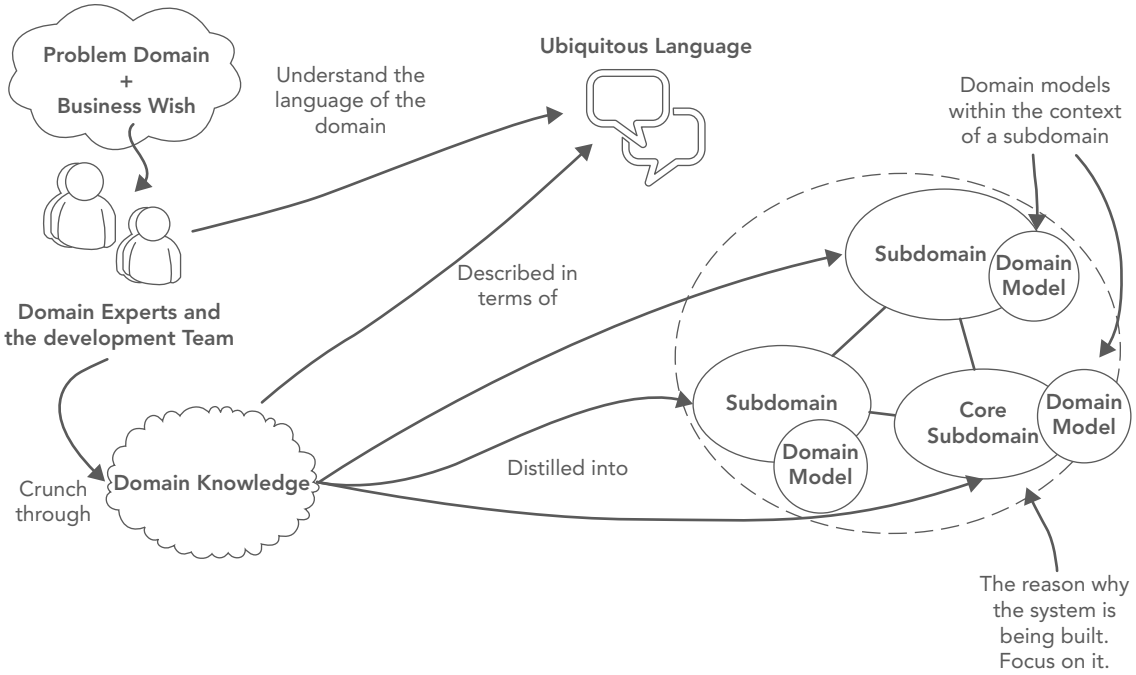

FIGURE 1-4: DDD patterns that are applicable to the problem space.

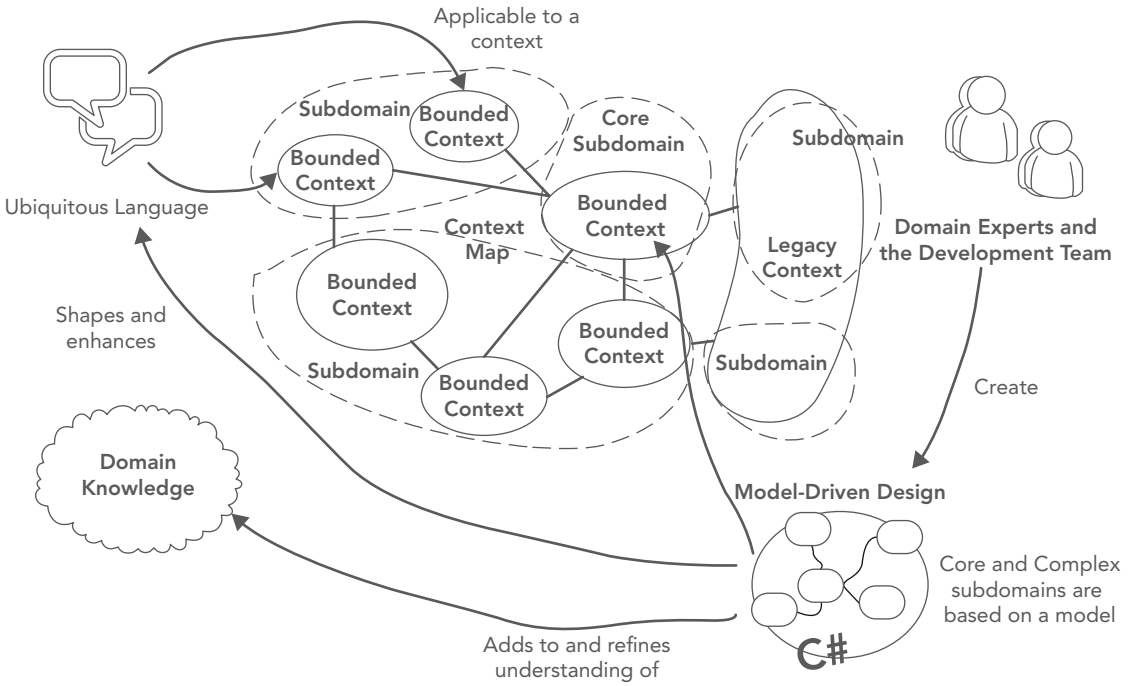

FIGURE 1-5: DDD patterns that are applicable to the solution space.

THE PRACTICES AND PRINCIPLES OF DOMAIN-DRIVEN DESIGN

Whilst there are many patterns of DDD, there are a number of practices and guiding principles that are key to success with its philosophy. These key principles, which form the essence of DDD, are often missed as too much focus is placed upon the tactical design patterns that are used to create software models.

Focusing on the Core Domain

DDD stresses the need to focus the most effort on the core subdomain. The core subdomain is the area of your product that will be the difference between it being a success and it being a failure. It's the product's unique selling point, the reason it is being built rather than bought. The core domain is the area of the product that will give you a competitive advantage and generate real value for your business. It is vital that all of the team understand what the core domain is.

Learning through Collaboration

DDD stresses the importance of collaboration between the development teams and business experts to produce useful models to solve problems. Without this collaboration and commitment from the business experts, much of the knowledge sharing will not be able to take place, and development teams will not gain deeper insights into the problem domain. It is also true that, through collaboration and knowledge crunching, the business has the opportunity to learn much more about its domain.

Creating Models through Exploration and Experimentation

DDD treats the analysis and code models as one. This means that the technical code model is bound to the analysis model through the shared UL. A breakthrough in the analysis model results in a change to the code model. A refactoring in the code model that reveals deeper insight is again reflected in the analysis model and mental models of the business. Breakthroughs only occur when teams are given time to explore a model and experiment with its design. Spending time prototyping and experimenting can go a long way in helping you shape a better design. It can also reveal what a poor design looks like. Eric Evans suggests that for every good design there must be at least three bad ones, this will prevent teams stopping at the first useful model.

Communication

The ability to effectively describe a model built to represent a problem domain is the foundation of DDD. This is why, without a doubt, the single most important facet of DDD is the creation of the UL. Without a shared language, collaboration between the business and development teams to solve problems would not be effective. Analysis and mental models produced in knowledge-crunching sessions between the teams need a shared language to bind them to a technical implementation. Without an effective way to communicate ideas and solutions within a problem domain, design breakthroughs cannot occur.

It is the collaboration and construction of a UL that makes DDD so powerful. It enables a greater understanding of the problem domain (for the business and the development team) and more

effective communication. These key values have a massive impact on projects because while technical frameworks and methodologies are important, DDD places as much, if not, greater importance on the analysis and understanding of the problem domain that ultimately makes software products successful.

Understanding the Applicability of a Model

Each model that is built is understood within the context of its subdomain and described using the UL. However, in many large models, there can be ambiguity within the UL, with different parts of an organization having different understandings of a common term or concept. DDD addresses this by ensuring that each model has its own UL that is valid only in a certain context. Each context defines a linguistic boundary; ensuring models are understood in a specific context to avoid ambiguity in language. Therefore a model with overlapping terms is divided into two models, each clearly defined within its own context. On the implementation side, strategic patterns can enforce these linguistic boundaries to enable models to evolve in isolation. These strategic patterns result in organized code that is able to support change and rewriting.

Constantly Evolving the Model

Any developer working on a complex system can write good code and maintain it for a short while. However, without synergy between the source code and the problem domain, continued development will likely end up in a codebase that is hard to modify, resulting in a BBoM. DDD helps with this issue by placing emphasis on the team to continually look at how useful the model is for the current problem. It challenges the team to evolve and simplify complex models of domains as and when it gains domain insights. DDD is still no silver bullet and requires dedication and constant knowledge crunching to produce software that is maintainable for years and not just months. New business cases may break a previously useful model, or may necessitate changes to make new or existing concepts more explicit.

POPULAR MISCONCEPTIONS OF DOMAIN-DRIVEN DESIGN

You can think of DDD as a development philosophy; it promotes a new domain-centric way of thinking. It is the learning process, not the end goal, which is the greatest strength of DDD. Any team can write a software product to meet the needs of a set of use cases, but teams that put time and effort into the problem domain they are working on can consistently evolve the product to meet new business use cases. DDD is not a strict methodology in itself but must be used with some form of iterative software project methodology to build and evolve a useful model.

Tactical Patterns Are Key to DDD

DDD is not a book on object-oriented design, nor is it a code-centric philosophy or a patterns language. However, if you search the web for articles on DDD, you would be mistaken for thinking that it is just a handful of implementation patterns as most articles and blogs on DDD focus on the modeling patterns. It is much easier for developers to see tactical patterns of DDD implemented in code rather than conversations between business users and teams on

a domain that they do not care about or do not understand. This is why DDD is sometimes mistakenly thought of as nothing more than a pattern language made up of entities, value objects, and repositories. You can, in fact, implement DDD without ever creating a rich domain model or using a repository. DDD is less about software design patterns and more about problem solving through collaboration.

Evans presents techniques to use software design patterns to enable models created by the development team and business experts to be implemented using the UL. However, without the practices of analysis, and collaboration, the coding implementation really means very little on its own. DDD is not code centric; its purpose is not to make elegant code. Software is merely an artifact of DDD.

DDD Is a Framework

DDD does not require a special framework or database. The model implemented in code follows a POCO (Plain Old C# Object) principle that ensures it is devoid of any infrastructural code so that nothing distracts from its domain-centric purpose. An object-oriented methodology is useful for constructing models, but it is by no means mandatory.

DDD is architecturally agnostic in that there is no single architectural style you must follow to implement it. A layered architectural style was presented in Evans's text, but this is not the only option. Architectural styles can vary because they should apply at the bounded context level and not the application level. A single product can include one bounded context that follows an event-centric architecture, another that utilizes a layered rich domain model, and a third that applies the active record pattern.

DDD Is a Silver Bullet

DDD can take a lot of effort, it requires an iterative development methodology, an engaged business, and smart developers. All software projects can benefit from the analysis practices of DDD such as distilling the problem domain as well as the strategic patterns such as isolating a code model that represents domain logic. However, not all require the tactical patterns of DDD to build a rich domain model. Trivial domains don't warrant the level of sophistication as they have little or no domain logic. For example, it would be a waste of time and costly to apply all of the patterns of DDD when creating a simple blogging application.

THE SALIENT POINTS

- Domain-Driven Design (DDD) is a development philosophy that is designed to manage the creation and maintenance of software written for complex problem domains.
- DDD is a collection of patterns, principles, and practices, which can be applied to software design to manage complexity.
- DDD has two types of patterns. Strategic patterns shape the solution, while tactical patterns are used to implement a rich domain model. Strategic patterns can be useful for any application but tactical patterns are only useful if your model is sufficiently rich in domain logic.

- Distillation of large problem domains into subdomains can reveal the core domain—the area of most value. Not all parts of a system will be well designed; teams must invest more time in the core subdomain(s) of a product.
- An abstract model is created for each subdomain to manage domain problems.
- A ubiquitous language is used to bind the analysis model to the code model in order for the development team and domain experts to collaborate on the design of a model. Learning and creating a language to communicate about the problem domain is the process of DDD. Code is the artifact.
- In order to retain the integrity of a model, it is defined within a bounded context. The model is isolated from infrastructure concerns of the application to separate technical complexities from business complexities.
- Where there is ambiguity in terminology for a model or multiple teams at work the model can be split and defined in smaller bounded contexts.
- DDD does not dictate any specific architectural style for development, it only ensures that the model is kept isolated from technical complexities so that it can focus on domain logic concerns.
- DDD values a focus on the core domain, collaboration, and exploration with domain experts, experimentation to produce a more useful model, and an understanding of the various contexts in play in a complex problem domain.
- DDD is not a patterns language, it is a collaboration philosophy focused on delivery, with communication playing a central role.
- DDD is a language- and domain-centric approach to software development.