Practicing Domain-Driven Design

Practical advice for teams implementing the development philosophy of Domain-Driven Design

With examples in C# .NET

Scott Millett

Principles, Patterns and Practices of Domain-Driven Design

Practical advice for teams implementing the development philosophy of Domain-Driven Design. With code examples in C# .NET.

Scott Millett

This book is for sale at http://leanpub.com/Practicing-DDD

This version was published on 2014-01-14

Tweet This Book!

Please help Scott Millett by spreading the word about this book on Twitter!

The suggested hashtag for this book is #Practicing-DDD.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#Practicing-DDD

Contents

About The Author

Scott Millett is an enterprise software architect working in London for Wiggle.co.uk, an e-commerce company specializing in cycle and triathlete sports. He has been working with .NET since version 1.0 and was awarded the ASP.NET MVP in 2010 and again in 2011. When not writing about or working with .NET he can be found relaxing and enjoying the music at Glastonbury and many of the major music festivals in the UK during the summer. If you would like to talk to Scott about the book, anything .NET, or the British music festival scene, feel free to write to him by email at scott@elbandit.co.uk, or by giving him a tweet @ScottMillett.

Scott's Books:

- Real World .NET, C#, and Silverlight: Indispensible Experiences from 15 MVPs (Chapter 13 BDD) (Wrox, 2011)
- Pro Agile .NET Development with SCRUM (Apress, 2011)
- Professional ASP.NET Design Patterns (Wrox, 2010)
- Professional Enterprise .NET (Wrox, 2009)
- NHibernate with ASP.NET Problem Design Solution (Wrox, 2009)

Introduction

Why You Should Read This Book

Writing software is easy, sorry writing greenfield software is easy. When it comes to modifiying code written by other developers or code you wrote six months ago, it can be a bit of a bore at best and a nightmare at worst. The software works but you aren't sure exactly how. It contains all the right frameworks, patterns and has been created using an agile approach, yet introducing new features into the code base is harder than it should be. Even business experts are of no use as the code bears no resemblance to the language they use. Working on such systems becomes a chore leaving developers frustrated and devoid of any coding pleasure.

Domain-Driven Design (DDD) is a process that aligns your code with the reality of your problem domain. As your product evolves, adding new features is as easy as it was in the good old days of greenfield development. Where DDD understands the need for software patterns, principles, methodologies and frameworks, it values developers and domain experts working together to understand domain concepts, policies and logic equally. With a greater knowledge of the problem domain, and a synergy with the business, developers are more likely to build software that is more readable and easier to adapt for future enhancement.

Following the DDD philosophy will give developers the knowledge and skills they need to tackle large or complex business systems in an effective manner. Future enchancement requests won't be met with an air of dread and developers will no longer have stigma attached to the legacy application. In fact the term 'legacy' will be recategorised in a developers mind as meaning; A system that continues to give value for the business.

Who Should Read This Book

This short book introduces the main themes behind Domain-Driven Design (DDD), its practices and principles along with my experiences and interpretation of the philosophy. It is intended to be used as a learning aid for those interested in or are starting out with the philosophy. It is not a replacement for "Domain-Driven Design: Tackling Complexity in the Heart of Software" by Eric Evans (Addison-Wesley Professional, 2003). Instead it takes the concepts introduced by Evans and distills them into simple straightforward prose, with practical examples in order for any developer to get up to speed with the philosophy before going on to study the subject in more depth.

This book is based on my experiences with the subject matter so you may not always agree with it if you are a seasoned DDD practitioner, but I do hope you still get something out of it. Either way please send me feedback on the book and let me know if you need any area explored in more detail or if I have not succeeded in my goal of explaining concepts simply and concisely.

I look forward to receiving your feedback - scott@elbandit.co.uk.

Domain-Driven Design In A Nutshell

This book will give you a thorough understanding about how you can apply the patterns and practices of Domain-Driven Design on your own projects but before we delve into the details lets take a birds eye view on the philosophy so you can get a sense of what DDD is really all about.

The Main Focus

- *Distilling The Problem*
  Complex problems are made up of parts that are not equal. Distil the problem area to discover the core problem sub domain.
- *Exploration And Learning Through Collaboration*
  Domain-Driven Design is about the process of learning a domain. In order to learn the team along with a domain expert must experiment with creating models to solve problems.
- *Constantly Refining and reshaping the language and the model*
  New cases may break a previous useful model or may require changes to existing or new concepts to be made explicit.
- *Defining Context Boundaries*
  A model should be bound to a context define by the language of the domain. Helps to support exploration and learning without effecting other teams.
- *Understanding The Core Domain*
  The majority of focus and effort should be placed in area of you domain that gives you the most advantage.

The Problem Space

Before you can develop a solution you must understand the problem. Domain-Driven Design emphasises the need to focus on the business problem domain; its terminology, the core reasons behind why the software is being developed and what success means to the business. The need for the development team to value domain knowledge just as much as technical expertise is vital to gain a deeper insight of the problem domain.

The figure below shows a high-level overview of the problem space of Domain-Driven Design that will be introduced in the first part of this book.

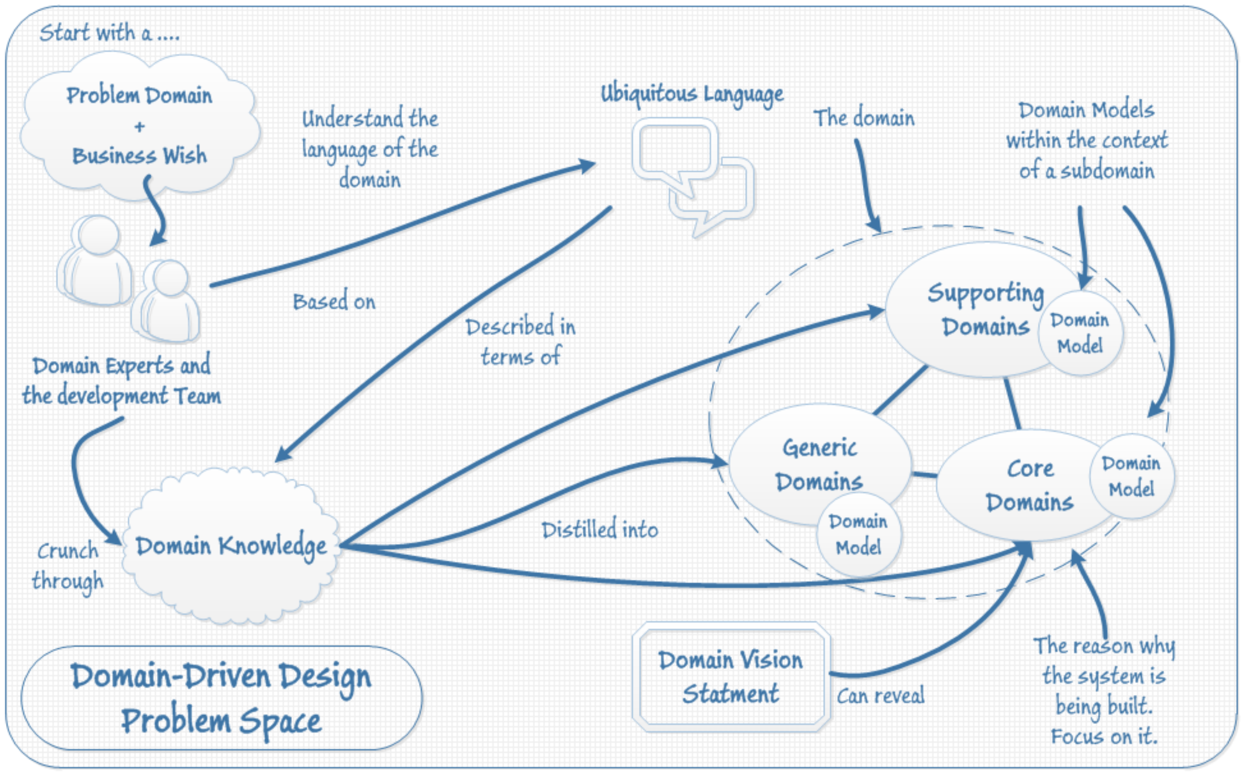
Start with a ....
Problem Domain + Business Wish
Understand the language of the domain
Ubiquitous Language
The domain
Domain Models within the context of a subdomain
Based on
Described in terms of
Domain Experts and the development Team
Supporting Domains
Domain Model
Generic Domains
Domain Model
Core Domains
Domain Model
Crunch through
Domain Knowledge
Distilled into
Domain-Driven Design Problem Space
Domain Vision Statment
Can reveal
The reason why the system is being built. Focus on it.

Figure I-1: A blueprint of the problem space of Domain-Driven Design.

The Solution Space

With a sound understanding of the problem domain strategic patterns of DDD can help you to implement a technical solution in synergy with the problem space. Patterns enable core parts of your system that are crucial to the success of the product to be protected from the generic areas. Isolating integral components allows them to be modified without having a rippling effect throughout the system.

Core parts of your product that are sufficiently complex or will frequently change should be based on a model. The tactical patterns of DDD along with Model-Driven Design will help you to create a model. A model that is in synergy with the problem domain will enable your software to be adaptable and understood by other developers and business experts.

The figure below shows a high-level overview of the solution space of Domain-Driven Design that will be introduced in the first part of this book.

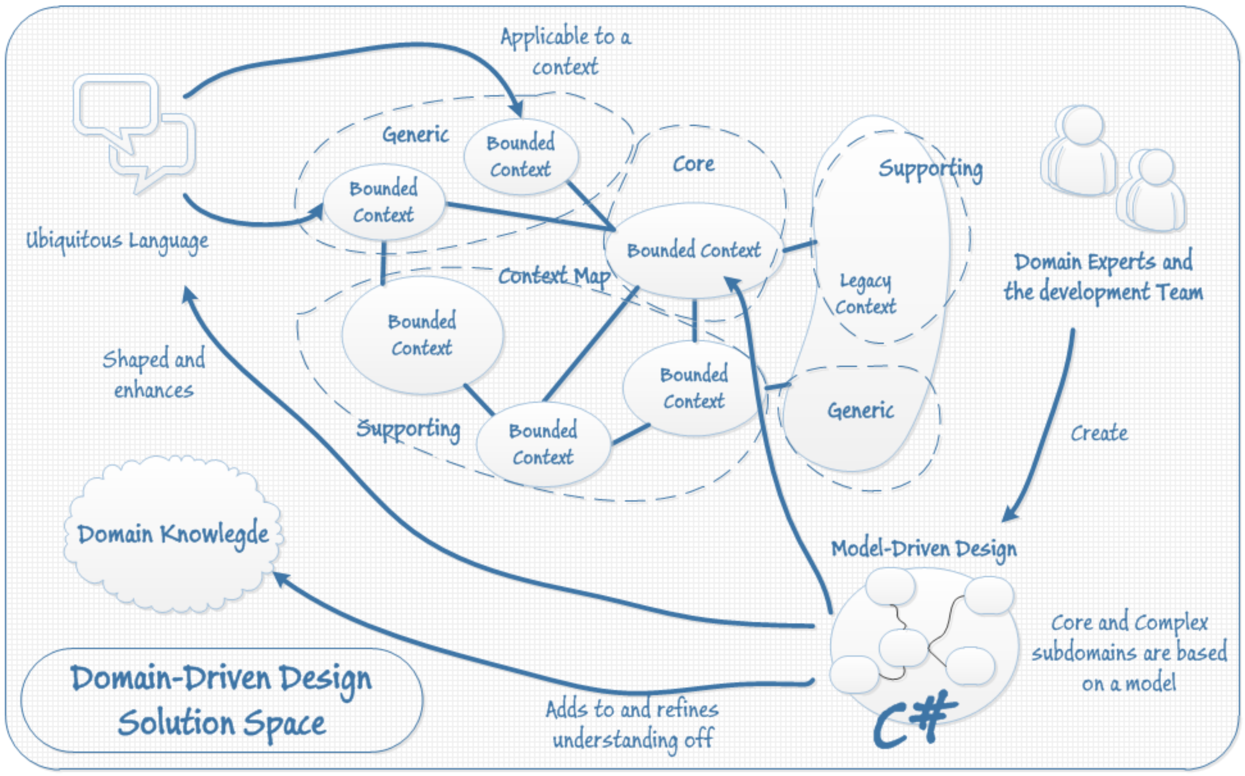

Figure I-2: A blueprint of the solution space of Domain-Driven Design.

How This Book Is Structured

This book is a draft copy of the Wrox Book "Principles, Patterns and Practices of Domain-Driven Design" by Scott Millett available from http://www.amazon.com/Professional-Domain-Driven-Design-Patterns/dp/1118714709/

The Principles and Practices of Domain-Driven Design

Part I of this book is an introduction to the principles and practices of Domain-Driven Design.

Covered in Part I:

- Chapter 1: The opening section of the book introduces you to the philosophy of DDD. You will learn why DDD is needed when dealing with the creation of large complex systems and how it can help you to avoid an unmanageable code base.
- Chapter 2: Distilling the problem space to reveal what is core.
- Chapter 3: The importance of the collaboration with Domain Experts to gain domain knowledge is discussed next. Domain knowledge will enable you to understand the most important areas of the application you are creating and where to spend the most time and effort.

- Chapter 4: The benefit of creating a shared Ubiquitous Language is then explored. The idea of a shared language is core to DDD and underpins the philosophy. A language describing the terms and concepts of the domain, which is created by both the development team and the business experts, is vital to aid communication on complex systems
- Chapter 5: The Domain Model.
- Chapter 6: Context. Strategic coding patterns designed to protect the integrity of core parts of the domain are presented after knowledge of the domain is gained along with patterns to aid team working.
- Chapter 7: Context Mapping. Understanding the relationships between different contexts in an application.
- Chapter 8: Application Architecture.
- Chapter 9: Common problems. The last section of part I describes when not to use Domain-Driven Design, and why it's just as important as knowing how to implement it. Simple systems require straight forward solutions. The end of this part looks at why applying DDD to simple problems can lead to overdesigned systems and needless complexity.
- Chapter 10: Getting Started with Domain-Driven Design.

Strategic Patterns: Communicating Between Bounded Contexts

Part II shows you how to integrate bounded contexts. Details on the options open for architecting Bounded Contexts. Code examples detailing how to integrate with legacy applications. Techniques for communicating across Bounded Contexts.

- Chapter 11: Introduction To Bounded Context Integration
- Chapter 12: Integrating with Messaging
- Chapter 13: Integrating with Open Host
- Chapter 14: Integrating with Legacy Contexts Via An Anti-Corruption Layer

Part II is available in the Wrox Book "Principles, Patterns and Practices of Domain-Driven Design" by Scott Millett available http://www.amazon.com/Professional-Domain-Driven-Design-Patterns/dp/1118714709/

Tactical Patterns: For Effective Domain Models

Part III focuses on the implementation of the tactical patterns of Domain-Driven Design and how to create an effective domain model. I show you implementations of the patterns contained within Eric's book along with some best practices to aid you with Model-Driven Design.

- Chapter 15: Introducing the Domain Model Building Blocks
- Chapter 16: Aggregates

- Chapter 17: Entities and Value Objects and Domain Services
- Chapter 18: Domain_Events
- Chapter 19: Life_Cycle_Patterns
- Chapter 20: Application Service Patterns
- Chapter 21: CQRS
- Chapter 22: Event Sourcing
- Chapter 23: DDD Best Practices

Part III is available in the Wrox Book "Principles, Patterns and Practices of Domain-Driven Design" by Scott Millett available http://www.amazon.com/Professional-Domain-Driven-Design-Patterns/dp/1118714709/

Other Books On Domain-Driven Design

Evans original text on Domain-Driven Design:

- Domain-Driven Design: Tackling Complexity in the Heart of Software by Eric Evans (Addison-Wesley Professional, 2003)

Other good books on Domain-Driven Design are:

- Applying Domain-Driven Design and Patterns: Using .Net by Jimmy Nilsson (Addison Wesley, 2006)
- Implementing Domain-Driven Design by Vaughn Vernon (Addison Wesley, 2013)
- Growing Object-Oriented Software, Guided by Tests by Steve Freeman and Nat Pryce (Addison Wesley, 2009)
- Specification by Example: How Successful Teams Deliver the Right Software by Gojko Adzic (Manning Publications, 2011)

Websites and groups on Domain-Driven Design:

- http://groups.yahoo.com/group/domaindrivendesign
- http://domaindrivendesign.org

The Principles and Practices of Domain-Driven Design

Chapter 1: What is Domain-Driven Design?

WHAT'S IN THIS CHAPTER?

- An introduction to the philosophy of Domain-Driven Design
- The value of applying Domain-Driven Design
- Why language and collaboration are the most important facets of Domain-Driven Design
- Why Domain-Driven Design is more than a collection of coding patterns
- How Domain-Driven Design Can help with the challenges of building software for complex domains

> **"It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change."** Charles Darwin

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 a software development approach designed to manage complex and large scale software products. At its heart it focuses on the creation of a shared language known as the Ubiquitous Language to efficiently and effectively describe a problem domain. This ever evolving Ubiquitous Language is the tool which enables development teams and the business to collaborate on building useful models of the problem domain which are used to solve business use cases. The model is represented in software using the same Ubiquitous Language and is at the heart of any technical implementation.

Many models may exist in a large problem domain and Domain-Driven Design stresses the need to isolate separate models in order to remove ambiguity in the Ubiquitous Language and to define boundaries of applicability for the models. Software design patterns in the book are presented to show how the mental model created by the development team and business experts can be bound to a technical implementation and that they can both evolve as one.

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.

The Value Of Domain-Driven Design

Domain-Driven Design focuses on the collaboration between a development team and business experts to create an understanding of a problem domain using a shared Ubiquitous Language to describe it. It is this collaboration and construction of a Ubiquitous Language 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 as they place far greater importance on analysis and understanding over technical frameworks and methodologies, which ultimately make software products successful.

A Focus On Collaboration

Domain-Driven Design stresses the importance of collaboration between the development teams and business users to produce useful models in order 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 true that through collaboration the business has time chance to model its own capabilities and learn much more about its problem domain.

A Focus On Language

Problems in software development are solved by language and the ability to effectively describe a problem domain forms the foundation of Domain-Driven Design. This is why without doubt the single most important facet of Domain-Driven Design is the creation of the ubiquitous language. 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 needs a shared language to bind it to a technical implementation. Without an effective way to communicate ideas and solutions within a problem domain design breakthroughs cannot occur.

A Focus On Building A Model For Core Domains

Domain-Driven Design treats the analysis and code models as one. This means that the technical code model is bound to the analysis mode through the shared Ubiquitous Language. 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.

If the core parts of your business software are sufficiently complex, then tactical patterns of DDD can help you create software of a high quality by basing the core domains (the parts of the system that will give you the most value) on an abstract model that reflects the real business subdomain. This model will enable the core parts of the system to be flexible enough to change when critical parts of the business change.

A Focus On Context

In many large problem domains there will be ambiguity within a shared language with different parts of an organization having different understandings of a common term or concept. Domain-Driven Design addresses this by ensuring models created by the development and business experts have a clear context and an understanding of the applicability of a model and its linguistic boundaries.

A Focus On Organization

On the analysis side of Domain-Driven Design strategic patterns enable the distillation of large problems domains in order to better manage communication and isolate models that have ambiguity within the shared language. In the implementation side strategic patterns can enforce these linguistic boundaries in order to enable models to evolve in isolation. These strategic patterns result in organized code which is able to support change and rewriting.

A Focus On The Bigger Picture

Domain-Driven Design understands the need to ensure that teams and the business are clear on the wider problem domain and how separate models and modules relate. Context maps help us to understand the bigger picture, they enable teams to understand what models exist and where there applicability boundaries lay. They reveal how different models interact and what data they exchange in order to fulfill business processes.

A Focus On What Is Important

Domain-Driven Design stresses the need to focus most effort on the core sub domain. The core domain is the area of your product that will be the difference between it being a success or a failure. It is the part of the product that is its unique selling point, the reason why it is being built rather then bought. It is the area of the product that will give you competitive advantage and generate really value for your business.

A Philosophy

Domain-Driven Design can be thought of a development philosophy, it promotes a new domain centric way of thinking. It is the learning process, not the end goal that is the greatest strength of Domain-Driven Design. Any team can write a software product to meet a problem, but teams that put time and effort in the problem domain that they are working on will consistently be able to evolve the product to meet new business use cases.

Not A Pattern Language

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 sometimes DDD is mistakenly thought of 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.

Not A Framework

Domain-Driven Design does not require any special framework or database. The model implemented in code follows a POCO (Plain Old Common Language Runtime Object) principle which 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 that must be followed in order to implement Domain-Driven Design. A layered architectural style was presented in Evans text but this is not the only option. Architectural styles can vary as they should apply at the bounded context level and not application. A single product could include a bounded context that followed a event centric architecture with another with a layered rich domain model pattern and a third using the active record pattern.

Not All About The Code

Contrary to populate belief Domain-Driven Design is not a book on object oriented design nor is it a code centric philosophy or indeed 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. Many of the coding patterns presented with Domain-Driven Design had been widely adopted before Evans 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. DDD is not code centric; its purpose is not to make elegant code. Software is merely an artifact of DDD.

Evans presents techniques to use software design patterns to enable models created by the development team and business experts to be implementation using the Ubiquitous Language. So without the process, analysis and collaboration the coding implementations really means very little on its own.

Not A Methodology

DDD is not a strict methodology in itself but must be used with some form of iterative software project methodology in order to evolve and build a useful model.

The Challenges Of Writing Large-Scale Complex Business Software

In order to understand how Domain-Driven Design can help with the design of software for a non trivial domain you must first understand the difficulties of creating and maintain software. By far, the most popular software architectural design pattern for business applications is the Big Ball of Mud (BBoM) pattern. BBoM, as defined by Brian Foote and Joseph Yoder.

> "A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle." Brian Foote and Joseph Yoder

Foote and Yoder use the term BBoM to describe an application that appears to have no distinguishable architecture (think big bowl of spaghetti versus a dish of layered lasagna). The issue with allowing software to dissolve into a BBoM becomes apparent when routine changes in work flow and small feature enhancements become a challenge to implement due to the difficulties in reading and understanding the existing code base. In his book, Eric Evans describes such systems as containing "code that does something useful but without explaining how."

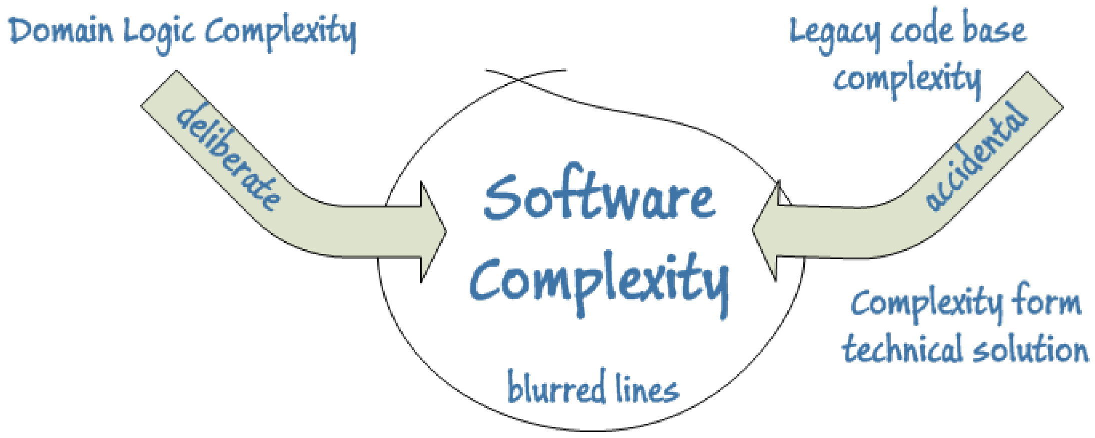

Figure 1-1: Complexity in code.

- TODO: Paragrah on the diagram above. Critical complexity is the domain logic, don't do Domain-Driven Design if no complexity in domain.

Code without a Common Language

A lack of focus on a shared language and knowledge of the problem domain results in a code base that works but does not reveal the intent of the business. This makes code bases very difficult to read and maintain as translations between the analysis model and business language to the code implementation and technical concepts can be costly.

Code without a binding to an analysis model that the business understands quickly conforms to the BBoM pattern meaning that no domain insight can possibly be discovered by the development team as the code base is focused on concepts that the business does not understand.

No Organization

As highlighted in figure 1-2 we have typically found that the initial incarnation of a system that resembles BBoM was fast to produce and a well-rounded success, but because there was little focus on business need, subsequent enhancements are troublesome. The code base lacks the required synergy with the business behavior to make change manageable. This sentiment is summed up rather well in the form of a tweet from Robert C Martin:

> "It doesn't take a lot of skill to get a program to work. The skill comes in when you have to *keep* it working." Robert C. Martin @unclebobmartin

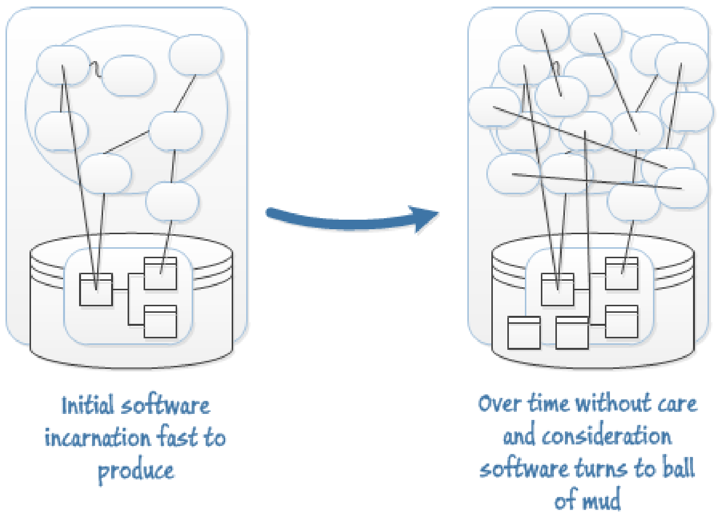
Initial software incarnation fast to produce
Over time without care and consideration software turns to ball of mud

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, often they can be buggy due to the unintelligible mess of the code base that developers have to deal with. Over time, the development team increasingly complains about the difficulty of working in such a mess. Even if resource is added to the project, velocity cannot be increased to a level that satisfies the business.

In the end, exacerbated by the situation, the request for the dreaded application rewrite is granted. Without due care and consideration, however, even the greenfield project can fall fowl 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 in terms of features and speed of delivery at the beginning but over time, even with additional investment in resource, did not see the sustained evolution of the product to meet the needs of the business.

A Lack Of Focus On The Business

Software projects fail when we don't understand the business domain we are working within well enough. Typing is not the bottleneck for delivering a product; coding is the easy part of development. Creating and keeping a useful software model of the domain that can be used to 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 software for it in order to solve problems within it.

Get Me Out Of This Big Ball Of Mud

So what is required to tackle the creation of a complex application? How can you ensure you are building solid systems that meet the *wishes* of the business, and that will work with you in the future, rather than working against you when the business needs to quickly alter a process or policy? As the enabler, you need to refocus your efforts when creating and designing software. You need to focus on the problem that the business is trying to solve with the application, not the next fashionable development practice or shiny new framework. You need a code base that contains a model of the domain in synergy with the business, constantly evolving as the business needs change. You need to align yourself with the vision of the software, clearly understanding what is core to the success of the application so that you can invest time and effort in the right places: You need some Domain-Driven Design.

How Domain-Driven Design Can Help?

The challenge of writing and enhancing software for large-scale complex systems goes far beyond any technical considerations. In fact, for complex domains, the technical challenges could be very simple. The development team should focus as much energy and effort on the domain logic as it does on the technical aspects. Teams should 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 their 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 the quality of code in the key areas of the system 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 will rot and turn into a big ball of mud, resulting in hard-to-maintain software.

Domain-Driven Design addresses the challenges of managing and designing large complex systems by firstly focusing efforts on aligning the software contours with the business capabilities and secondly revealing the core parts of a system and thus understanding where developers should put most effort. The alignment is accomplished by breaking down large problem domains into smaller more focused sub domains and within them smaller business contexts. The core parts of the system are found by distilling the problem domain to reveal the fundamental reason why the software is being produced and what will be key to its success.

Revealing What Is Important

Not all of a large software product needs be perfectly design and trying to do so would be a waste of efforts. DDD helps to reveal what is the driving force behind the product under development, what is the fundamental reason it is being built. With an understanding on the key parts of any product teams can focus their efforts and ensure that the part of a product which will give the most value is the part that they pay most attention to and where they focus quality.

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

Isolate And Organize Code

The ability to break down large and complex problem domains and focus the teamâ€™s efforts on the most valuable parts of a systems is known as the strategic side of Domain-Driven Design. This strategic design helps to prevent software evolving into a big ball of mud as components of the overall solution are able to evolve within well defined business contexts without having a negative rippling impact on other parts of the system. It also arms the team with the knowledge on what to invest more time and effort in as they understand what is important to the business.

Compare the diagram in figure 1-3 to figure 1-2. The diagram shows how the strategic patterns of Domain-Driven Design have been applied to the software in order to manage the large problem domain.

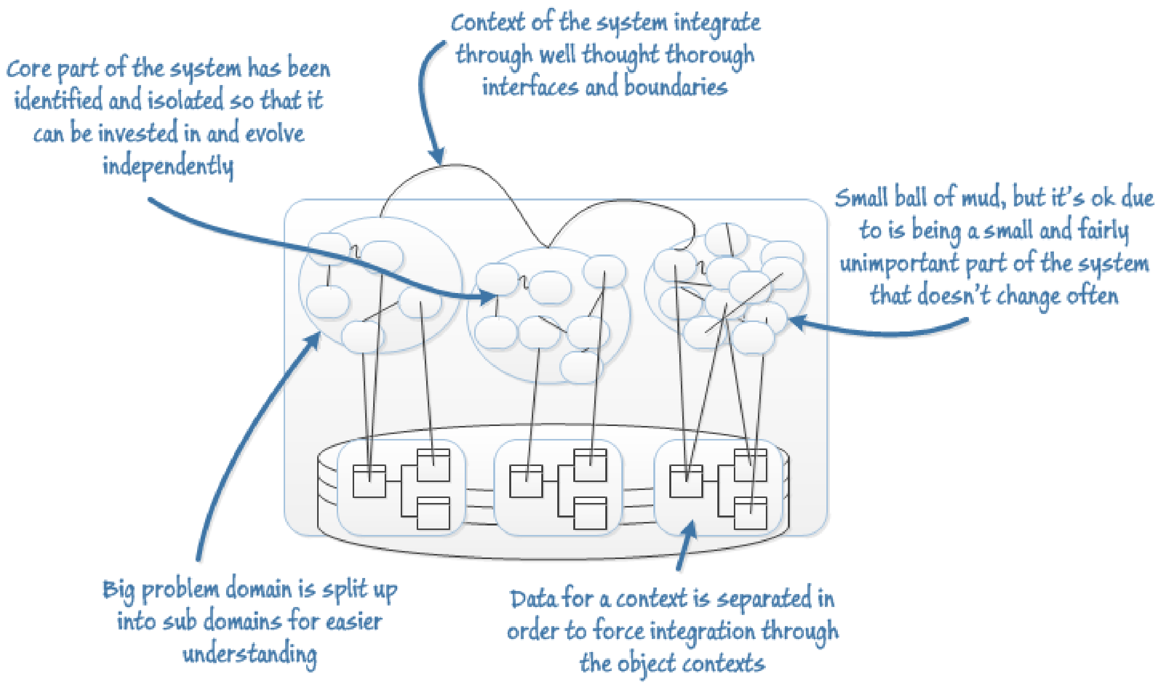
Context of the system integrate through well thought thorough interfaces and boundaries
Core part of the system has been identified and isolated so that it can be invested in and evolve independently
Small ball of mud, but it's ok due to is being a small and fairly unimportant part of the system that doesn't change often
Big problem domain is split up into sub domains for easier understanding
Data for a context is separated in order to force integration through the object contexts

Figure 1-3: Applying The Strategic Patterns Of Domain-Driven Design.

The Big Ball of Mud Is Not Always An Anti Pattern Not all parts of a large application will be designed perfectly - nor do they need to be. While it's not advisable to build an entire enterprise software stack following the big ball of mud pattern, you can however, 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 could be 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 deem the product to be a success and worthy of prolonged investment. Remember software is only a means to an end to support a business.

Model What Is Core

Once the key areas of the software are uncovered teams can leverage the tactical patterns of Domain-Driven Design to build a conceptual model, known as a domain model, of the core sub domain in order to provide a useful problem solving device for business use cases. This model is not a model of real life but more an abstraction build to satisfy the requirements of business use cases whilst still retaining the rules and logic of the business domain. Other sub domains that are not core to the success of the product or that are not as complex need not be based on a rich domain model and can instead utilize more procedural or data driven architectures.

System Maintainability

Any developer working on a complex system can write good code and maintain it for a short while. However, without a synergy between the code base and the problem domain continued development will likely end up in a code base that is hard to modify resulting in a big ball of mud. Domain-Driven Design helps with this issue by placing emphasis on the team to continually look at how useful there code is for the current problem and challenges them to evolve and simplify complex models of domains as and when they gain domain insights. Domain-Driven Design is still no silver bullet and requires dedication and constant knowledge crunching to produce software that is maintainable for years and not just months.

A Solution Space Vs A Problem Space

- TODO: Explain the different between solution space and problem space.

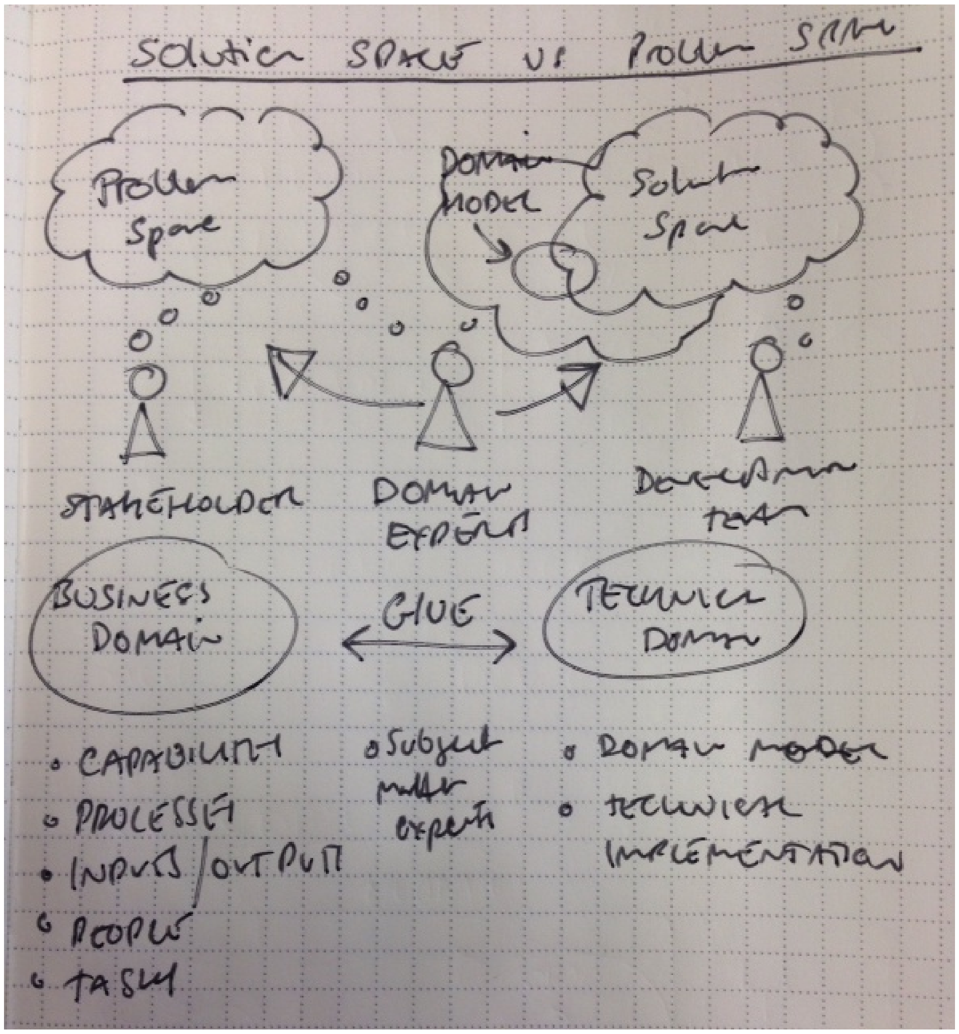

Figure 1-4: Applying The Strategic Patterns Of Domain-Driven Design.

The Salient Points

- Domain-Driven Design is a development philosophy that values teams understanding the domain they are writing software for above anything else.
- It is a collaboration philosophy focused on delivery with communication playing a central role.
- The creation of a shared language is vital for knowledge sharing and domain understanding by the development team and business experts.
- Domain-Driven Design is a language and domain centric approach.
- Domain-Driven Design Focuses on Language and communication within a defined context.
- Strategic patterns of Domain-Driven Design help to break down large problem domains.

- Tactical patterns of Domain-Driven Design help to model the core parts of the domain as a model.
- Not all parts of a system will be well designed; teams must invest more time in the core areas of a product.
- Domain-Driven Design is more problem solving through collaboration than code patterns language.
- Learning and creating a language to communicate about the problem domain is the process of Domain-Driven Design, code is the artifact.

Strategic Patterns: Communication Between Bounded Contexts

Part II is available in the Wrox Book “Principles, Patterns and Practices of Domain-Driven Design” by Scott Millett available http://www.amazon.com/Professional-Domain-Driven-Design-Patterns/dp/1118714709/

Tactical Patterns For Creating Effective Domain Models

Part III is available in the Wrox Book "Principles, Patterns and Practices of Domain-Driven Design" by Scott Millett available http://www.amazon.com/Professional-Domain-Driven-Design-Patterns/dp/1118714709/