THE ANATOMY OF DOMAIN-DRIVEN DESIGN

BY SCOTT MILLETT

DOMAIN-DRIVEN DESIGN IS A DEVELOPMENT APPROACH TO MANAGING SOFTWARE FOR COMPLEX DOMAINS

Domain-Driven Design is a language and domain-centric approach to software design for complex problem domains. It consists of a collection of patterns, principles and practices that will enable teams to focus on what is core to the success of the business while crafting software that manages complexity in both the technical and business spaces.

Complexity in software is the result of inherent domain complexity (essential) mixing with technical complexity (accidental).

Complexity from the domain is inherent

Complexity from the technical solution is accidental

How software for complex domains can become difficult to manage

1. Initial software incarnation fast to produce
2. Over time, without care and consideration, software turns into the pattern known as the "ball of mud"
3. It works but no one knows how. Change is risky and difficult to complete. Where technical complexity exists the best developers will spend time there and not in problem domain

FOCUS ON THE OPPORTUNITY AND COMPLEXITY OF THE CORE DOMAIN

In order to manage complexity in the solution space, developers need to conquer the problem space. Not all parts of a problem need perfect solutions. In order to reveal where most effort and expertise should be focused, large problem domains can be distilled. This enables the best developers to focus attention on areas of the problem domain that are key to the success of the product as opposed to the areas that offer the most exciting technical challenges.

Start from the Use Cases
A good place to start when trying to understand a new domain is by mapping out use cases. A use case lists the steps required to achieve a business outcome, including the interactions between users and systems.

Focus on the Most Interesting Conversations
Don't bore domain experts and business stakeholders by going through a list of requirements one item at a time. Start with the areas of the problem domain that keep the business up at night—the areas that will make a difference to the business and that are core for the application to be a success.

A Complex Business Domain

Facilitation Patterns

Engaged Development team and stakeholders

Ask Powerful Questions
What does good look like? What is the success criteria of this product? What will make it a worthwhile endeavour? What is the business trying to achieve? The questions you ask stakeholders and sponsors will go a long way toward your understanding of the importance of the product you are building and the intent behind it.

Visualise the Problem Domain
People often learn quicker by seeing visual representations of the concepts they are discussing. Sketching simple diagrams is a common visualization technique DDD practitioners use to enhance knowledge-crunching sessions and maximize their time with stakeholders and business experts.

Read the Product Vision
Understand the benefits that this project will realise, share the goals with the business stakeholders.

Employee Facilitation Patterns
Jeff Pattons' user story mapping, Alberto Brandolini's event storming techniques and Impact Mapping are three great ways to engage stakeholders and reveal the core of the product.

Consider allowing junior developers to sharpen their skills or outsource the development or integration of software for supporting subdomains

Don't be distracted by shiny technology. The core sometimes isn't the most technically challenging.

Supportive · Generic · Supportive · Generic · Core

This is what makes your software worth writing
- Attack complexity in the core opportunity.
- All interesting conversations will happen here.
- Apply the most effort here.
- Isolate the core domain from the rest of the problem
- Keep your wits about you, your core domain could change over time!

Generic subdomains can be satisfied by off the shelve packages, don't waste too much time here. This needs to be good enough.

SOLVE COMPLEX PROBLEMS BY USING MODELS

With the domain distilled and primary focus on the core domain a model can be produced to manage the complexity of the domain logic. A model is an abstraction of the problem domain used to solve business use cases. A model is discovered through the act of knowledge crunching business use case scenarios with domain experts. The domain model encapsulates complex business logic, closing the gap between business reality and code.

THE PROBLEM SPACE

PROBLEM DOMAIN · MORE FACILITATION PATTERNS · BUSINESS USE CASES · CONTEXT MAP

THE SOLUTION SPACE

Knowledge crunching is the art of distilling relevant information from the problem domain in order to build a useful model that can fulfil the needs of business use cases.

THE ANALYSIS MODEL — Domain Experts, Development Team

THE CODE MODEL

The Domain Model = Analysis Model + Code model + Language

Challenge Your Model
With each new business case and scenario your model will evolve. Don't become too attached as it's healthy to attack problems in a completely different way to reveal insights.

Don't Stop At Your First Good Idea
Many models must be rejected in order to ensure you have a useful model for the current use cases of a system.

The domain model is:
- An abstraction of reality – not a reflection of real life
- Designed to manage complexity for specific business cases.
- A single model that exists in code, language and written documentation and diagrams

DESIGN A MODEL IN COLLABORATION USING A UBIQUITOUS LANGUAGE

The development team and domain experts knowledge crunch to produce a model that satisfies the needs of the business use cases. Communication happens because of the ubiquitous language. Collaboration is key to creating a useful model and the ubiquitous language binds the abstract mental model to the underlying software model, enabling developers and domain experts to talk about things easily. It is the language that enables both the business and development teams to have meaningful communication about the software.

Obsess over this!!!

THE UBIQUITIOUS LANGUAGE
When modeling with stakeholders and domain experts, everyone should make a conscious effort to consistently apply a shared language rich in domain-specific terminology. This language must be made explicit and be used when describing the domain model and problem domain.

BUSINESS TERMINOLOGY
REMOVE business terms that are not useful to solving specific use cases even if they exist in reality
ADD business terms that can unlock key insights into the model
DOMAIN EXPERTS · BUSINESS COLLEAGUES

TECHNICAL TERMINOLOGY
REMOVE technical terms that distract from the core domain complexity and are not used by the domain experts
ADD concepts discovered in code that unlock deep insights into the domain
DEVELOPMENT TEAM · TECHNICAL COLLEAGUES

WRITE SOFTWARE THAT EXPLICITLY EXPRESSES THE MODEL

The ubiquitous language should be used in the code implementation of the model, with the same terms and concepts used as class names, properties, and method names. Continuous experimentation and exploration in the design of a model is where the power of DDD is realised.

START HERE

Focus on a single business use case at a time and model the various concrete scenarios for each use case.

Create a useful model that satisfies the needs of the use case. Don't be over ambitious and avoid perfectionism. The goal is not to model reality, your models should be inspired by aspects of reality.

Reveal hidden insights and simplify the model by exploring and experimenting with new ideas. You will understand more about the problem domain the more you play with it.

Don't stop modelling at the first useful model. Experiment with different designs to find a supple model and design breakthrough. Challenge your assumptions and look at things from a different perspective.

Apply tactical design patterns to model the rich domain behaviours and to ensure that the model is supple enough to adapt as new requirements surface.

Warning!; DDD is not a patterns language, don't fall into the trap of solely focusing on tactical code design patterns.

Isolate the model from infrastructure concerns and keep technical complexities separate from domain complexities. Use application services to model use cases and delegate to the domain model to solve.

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.

DIVIDE COMPLEX AND LARGE MODELS INTO SEPARATE BOUNDED CONTEXTS

Large or complex models are divided into bounded contexts where there is ambiguity in terminology, multiple teams are involved, where subdomains exists or where legacy code remains. Software that fails to isolate and insulate a model in a bounded context will often slip into the Ball of Mud pattern.

A domain model can grow in complexity over time to satisfy the different business use cases and different contexts of the business...

BUSINESS USE CASE · Knowledge crunching · WAREHOUSE CONTEXT · PROCUREMENT CONTEXT · SHIPPING CONTEXT

PRODUCT

In a single model, multiple teams will dilute the explicitness due to a lack of shared context.

In order to retain the integrity of a model, it is defined within a bounded context.

The context map
The context map details the relationships between the bounded contexts

PROCUREMENT CONTEXT — The product concept now exists in three different contexts this retains the models explicitness

SHIPPING CONTEXT

WAREHOUSE CONTEXT

Small ball of mud, but it's okay as it is low complexity and is isolated to prevent corruption to other contexts

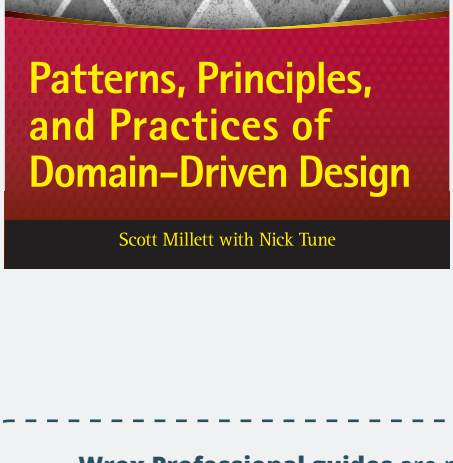
An anti corruption layer protects your models from balls of mud, by keeping the model pure

Third Party system that we can't control so we define it in its own context

LEARN MORE ABOUT DOMAIN-DRIVEN DESIGN

PATTERNS, PRINCIPLES, AND PRACTICES OF DOMAIN-DRIVEN DESIGN

BY SCOTT MILLETT & NICK TUNE



Build solutions for complex business problems more effectively with Domain-Driven Design

This book distills the ideas and theories of the Domain-Driven Design (DDD) philosophy into a practical playbook that you can leverage to simplify application development for complex problem domains. A focus is placed on the principles and practices of decomposing a complex problem space as well as the implementation patterns and best practices for shaping a maintainable solution space. You will learn how to build effective domain models through the use of tactical patterns and how to retain their integrity by applying the strategic patterns of DDD. Full end-to-end coding examples demonstrate techniques for integrating a decomposed and distributed solution space while coding best practices and patterns advise you on how to architect applications for maintenance and scale.

- Offers a thorough introduction to the philosophy of DDD for professional developers
- Simplifies the theories of Domain-Driven Design into practical principles and practices
- Includes masses of code and examples of concepts in action that other books have only covered theoretically
- Covers the patterns of CQRS, Messaging, REST, Event Sourcing and Event-Driven Architectures
- Ideal for developers using Java, Ruby, and other languages who want to learn common DDD implementation patterns
- Code examples presented in C# demonstrating concepts that can be applied in any language