THE ANATOMY OF DOMAIN DRIVEN DESIGN

BY SCOTT MILLETT

DDDEUROPE.COM

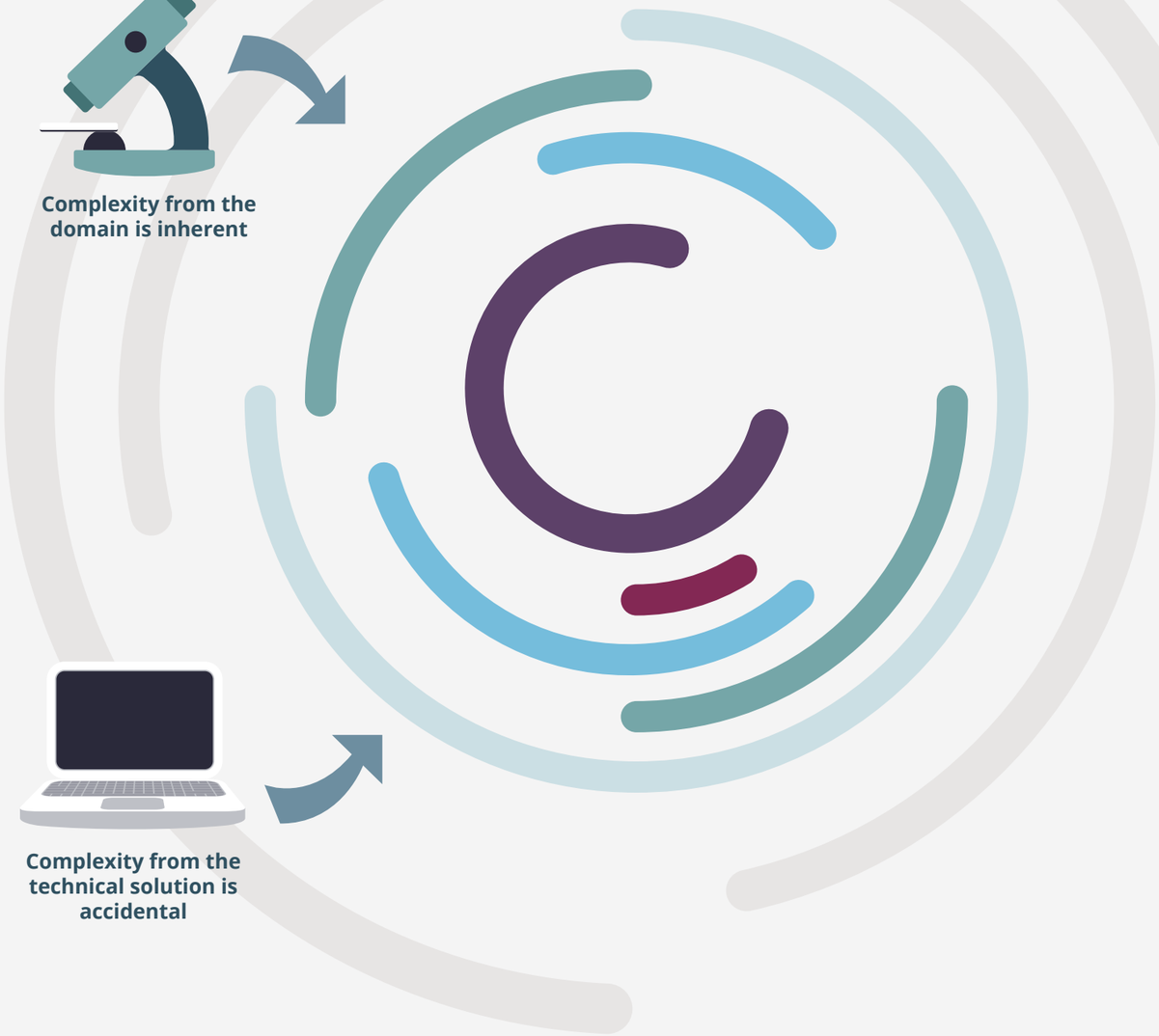
DOMAIN-DRIVEN DESIGN IS A DEVELOPMENT APPROACH TO MANAGING SOFTWARE FOR COMPLEX DOMAINS

Domain-Driven Design is a language and domain-centric approach to software design for complex problem domains. The term was coined by Eric Evans in his seminal book “Domain-Driven Design: Tackling Complexity in the Heart of Software”. 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.

Eric Evans
Domain-Driven Design:
Tackling Complexity in the Heart of Software

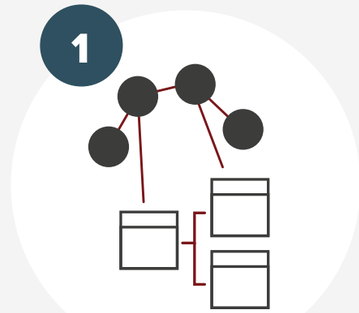
Complexity in software is the result of inherent domain complexity (essential) mixing with technical complexity (accidental).

Complexity from the domain is inherent

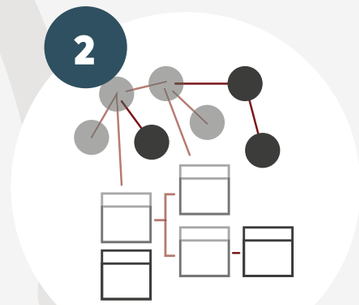


Complexity from the technical solution is accidental

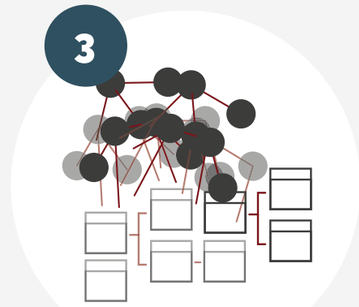
How software for complex domains can become difficult to manage



Initial software incarnation fast to produce



Over time, without care and consideration, software turns into the pattern known as the “ball of mud”



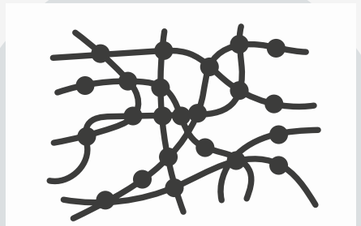
It works but no one knows how. Change is risky and difficult to complete. Where technical complexity exists the best developers will spend time there and not in problem domain

SOLVE COMPLEX PROBLEMS BY USING MODELS

A Domain Model represents a view, not the reality, of the problem domain, it exists only to meet the needs of business use cases. The various expressions of the model - code, diagram, docs - are bound by the same language. Its usefulness comes from its ability to represent complex logic and policies in the domain to solve business use cases. The model contains only what is relevant to solve problems in the context of the application being created. It needs to constantly evolve with to keep itself useful and valid in the face of new uses cases.

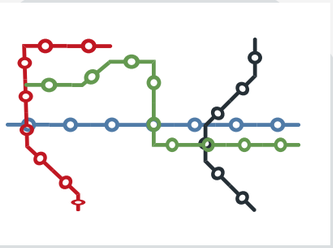
An example of a domain model...

THE USE CASE
(THE PROBLEM)



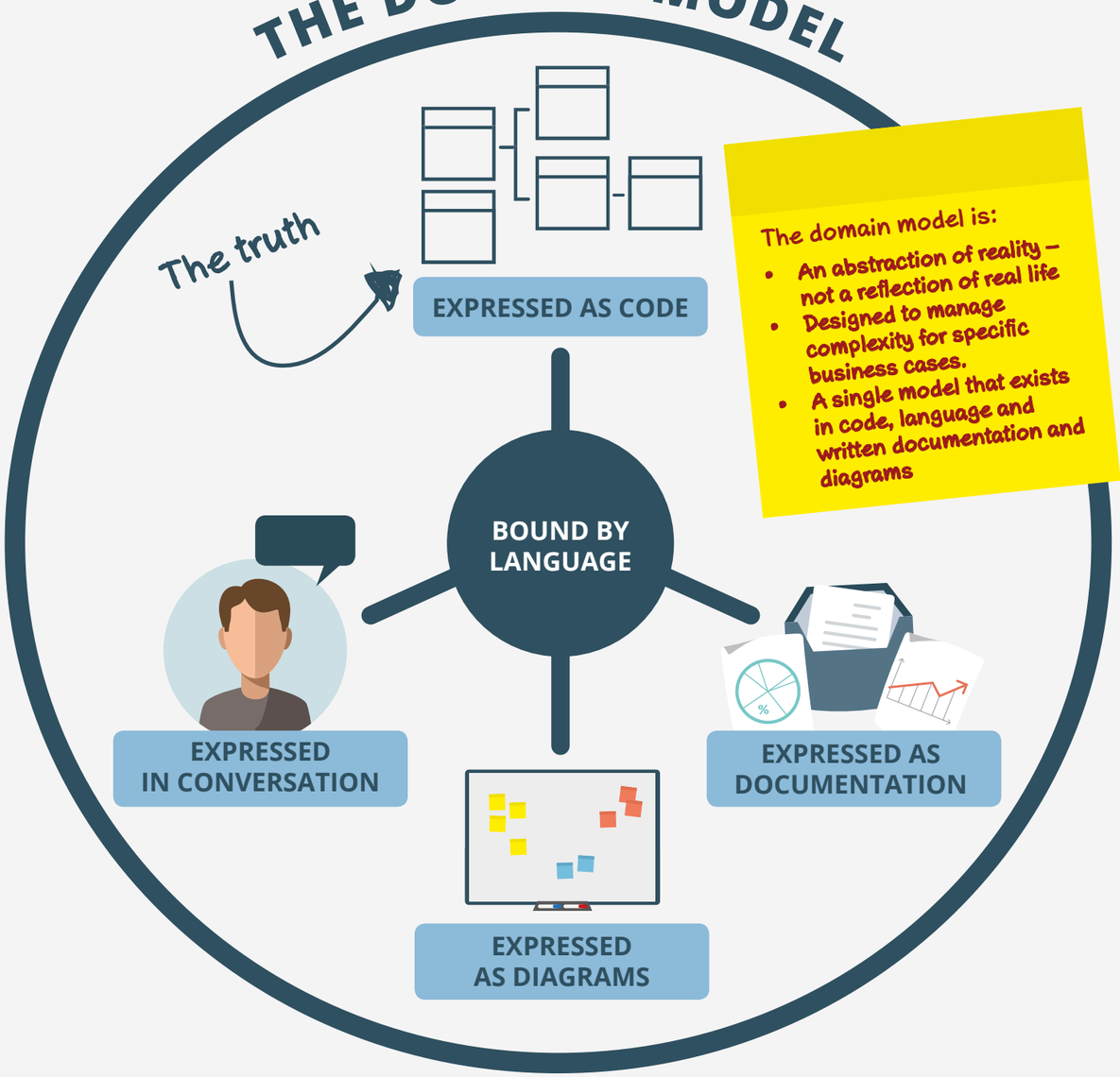
THE DOMAIN
(THE LONDON TUBE NETWORK)

The model solves the use case simply and effectively



THE DOMAIN MODEL
(THE LONDON TUBE MAP)

The model is an abstraction of the real domain, it bears little resemblance to the distance between stations



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.

Challenge Your Model

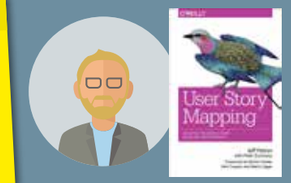
With each new business case and scenario your model will evolve. Don't become too attached as it's healthy to attack problems in a completely different way to reveal insights.

Don't Model Real LIfe

Model only what is relevant to solve use cases.

EMPLOY KNOWLEDGE CRUNCHING TECHNIQUES TO PRODUCE EFFECTIVE MODELS

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. Creating a useful model is a collaborative experience; however, business users can also find it tiring and can deem it unproductive. Business users are busy people. To make your knowledge crunching session fun and interactive, you can introduce some facilitation games and other forms of requirement gathering to engage your audience.



Jeff Patton
User Story Mapping

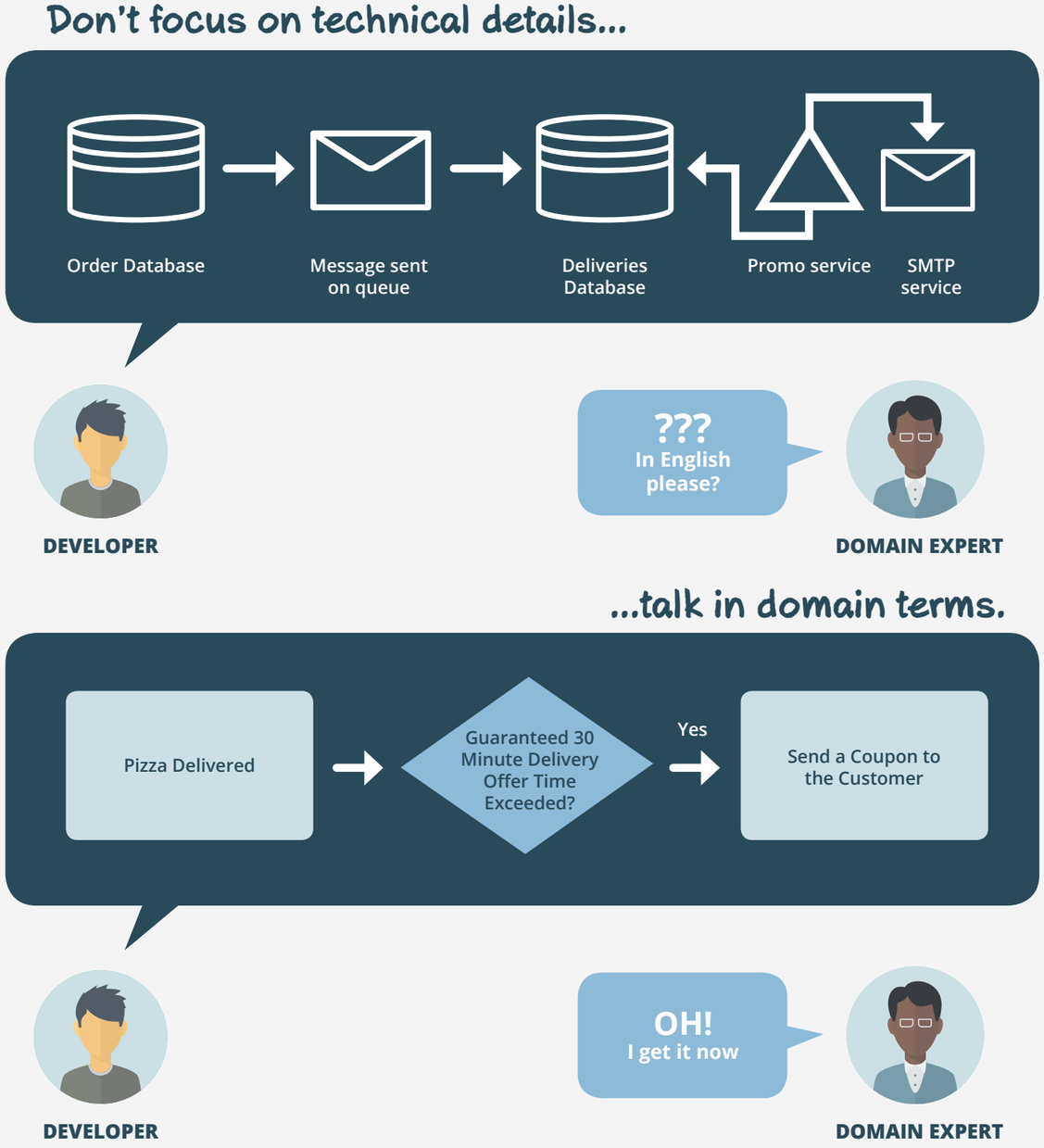
Alberto Brandolini
Event Storming

Eric Evans
DDD Whirpool

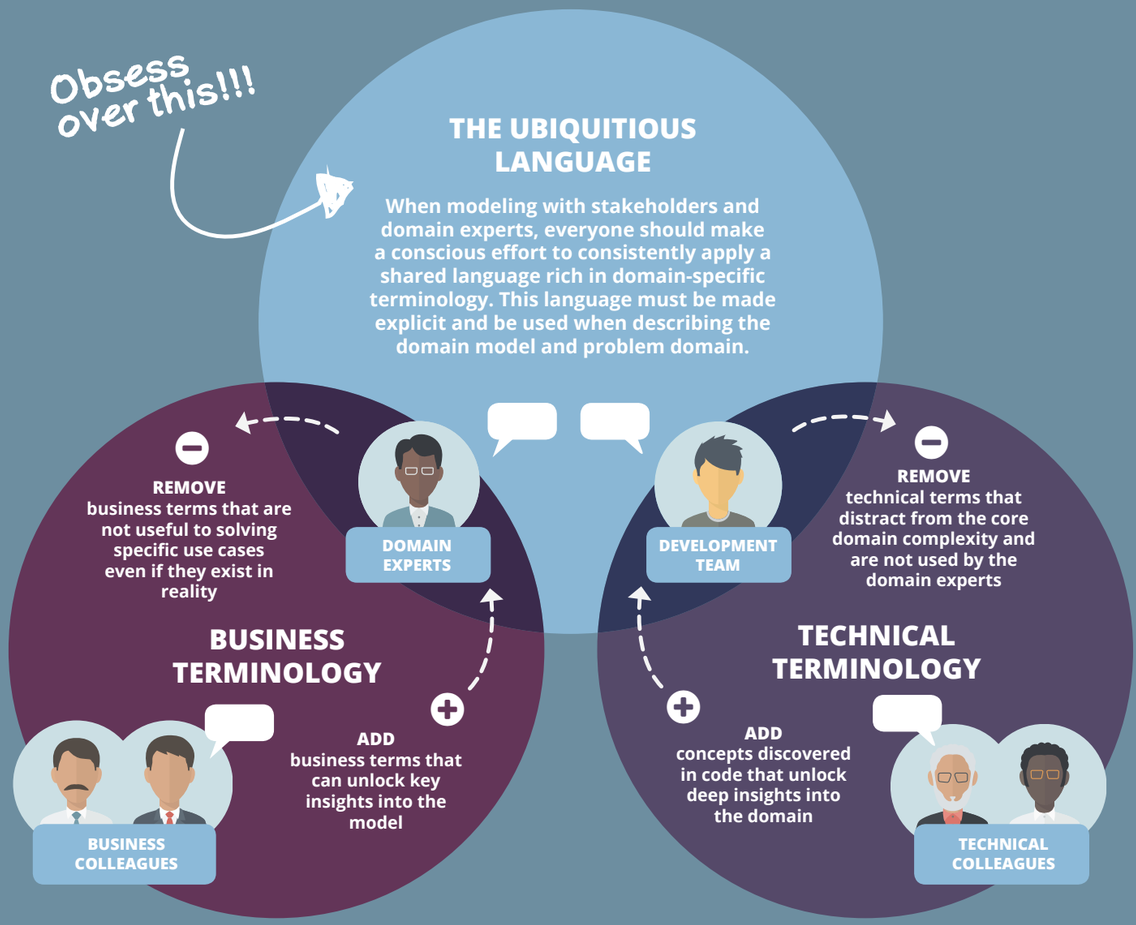
- "Ignorance is the single greatest impediment to throughput" - Dan North
- "The Critical Complexity of most software projects is in understanding the domain itself" - Eric Evans
- Knowledge is gained around whiteboards, water coolers, brainstorming, and prototyping in a collaborative manner, with all members of the team at any time of the project.
- Domain experts are the subject matter experts of the organization. They are anyone who can offer insight into the problem domain (users, product owners, business analysts, other technical teams).
- Your stakeholders will give you the requirements of your application but they may not be best placed to answer detailed questions of the domain. Utilize domain experts when modeling core or complex areas of the problem domain.
- Engage with your domain experts on the most important parts of a system. Don't simply read out a list of requirements and ask them to comment on each item.
- Plan to change your model; don't get too attached as a breakthrough in knowledge crunching may render it obsolete.
- Drive knowledge crunching around the most important uses case of the system. Ask the domain experts to walk through concrete scenarios of system use cases to help fill knowledge gaps.
- Ask powerful questions and learn the intent of the business. Don't simply implement a set of requirements, instead actively engage with the business; work with them, not for them.
- Visualize your learning with sketches and event storming techniques. Visualizing a problem domain can increase collaboration with the business experts and make knowledge-crunching sessions fun.
- Use BDD to focus on the behavior of the application and focus domains experts and stakeholders around concrete scenarios. BDD is a great catalyst for conversations with the domain experts and stakeholders. It has a template language to capture behavior in a standard and actionable way.
- Experiment in code to prove the usefulness of the model and to give feedback on the compromises that a model needs to make for technical reasons.
- Look at existing processes and models in the industry to avoid trying to reinvent the wheel and to speed up the gaining of domain knowledge.
- Find out what you don't know, identify the team's knowledge gaps early then activate deliberate discovery. Eliminate unknown unknowns and increase domain knowledge early.
- Leverage Eric Evans' Model Exploration Whirlpool when you need guidance on how to explore models. The activities in the whirlpool are particularly helpful when you are having communication breakdowns, overly complex designs, or when the team is entering an area of the problem domain of which they don't have much knowledge.

DESIGN A MODEL IN COLLABORATION USING A UBIQUITOUS LANGUAGE

Software projects fail due to poor communication and the overhead of translation between domain and technical terminology.



A Ubiquitous Language minimizes the cost of translation and binds all expressions to the code model also known as the true model. A shared language also helps collaborative exploration when modelling, which can enable deep insights into the domain.

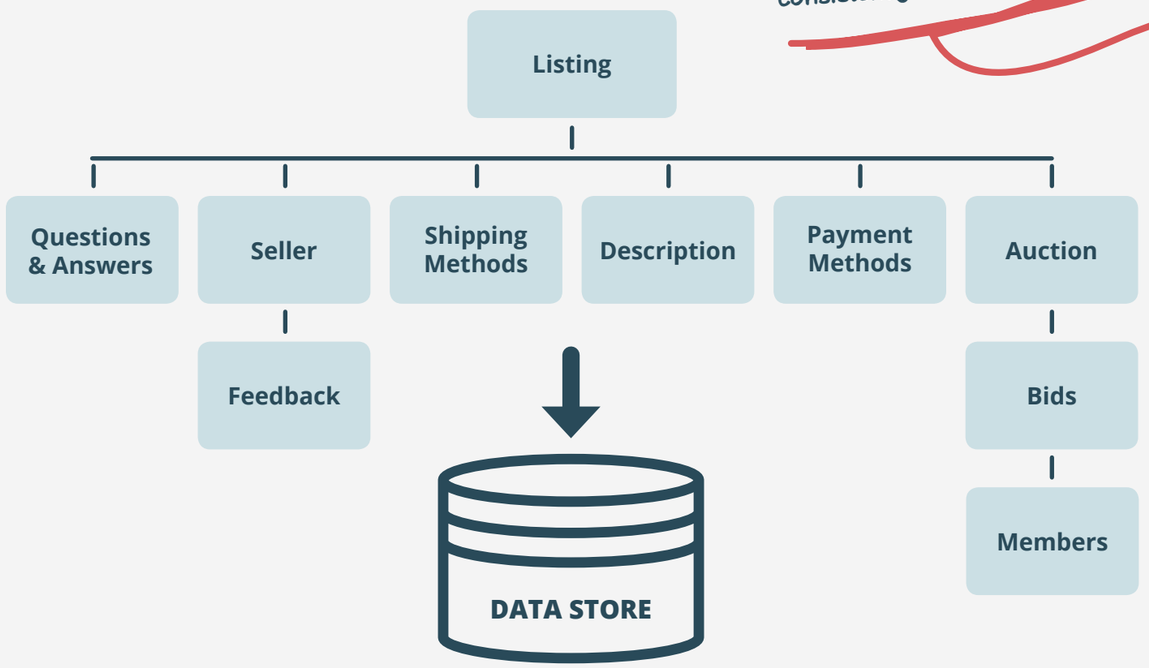


HOW TO IMPLEMENT YOUR MODEL

There are many tactical patterns presented in Eric Evan's book that will guide you toward creating a flexible and maintainable domain model. Many DDD practitioners find aggregates a useful tactical modeling heuristic. The aggregate pattern suits both an object oriented and functional programming style.

Problem

A single object graph may closely relate to the real domain but it does not make for an effective model. Treating the model as a single consistency boundary in a collaborative domain can lead to conflict for changes that are completely unrelated. Such as in the auction example below asking a question while someone is trying to place a bid.

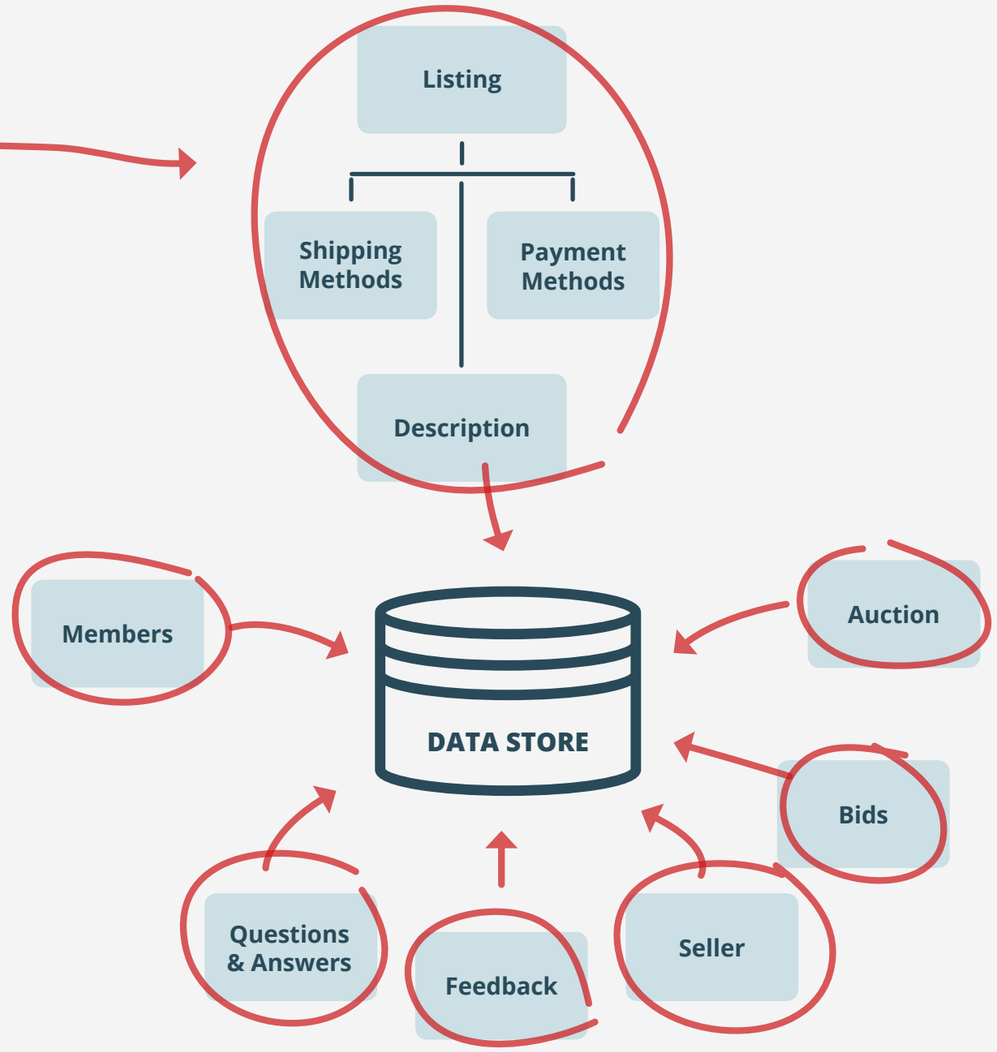


I can't stress enough that DDD does not dictate any special architectural style or require any special design patterns for development. Don't get too hung up on the tactical patterns. Make sure you understand the problem domain and isolate your domain code. You will find that even the aggregate pattern is optional.

The area of domain model implementation has evolved greatly in the decade that has passed since Eric Evans original wrote about DDD. DDD is technically agnostic, therefore it's important to understand that the implementation tactics for building domain models should remain flexible and open to innovation.

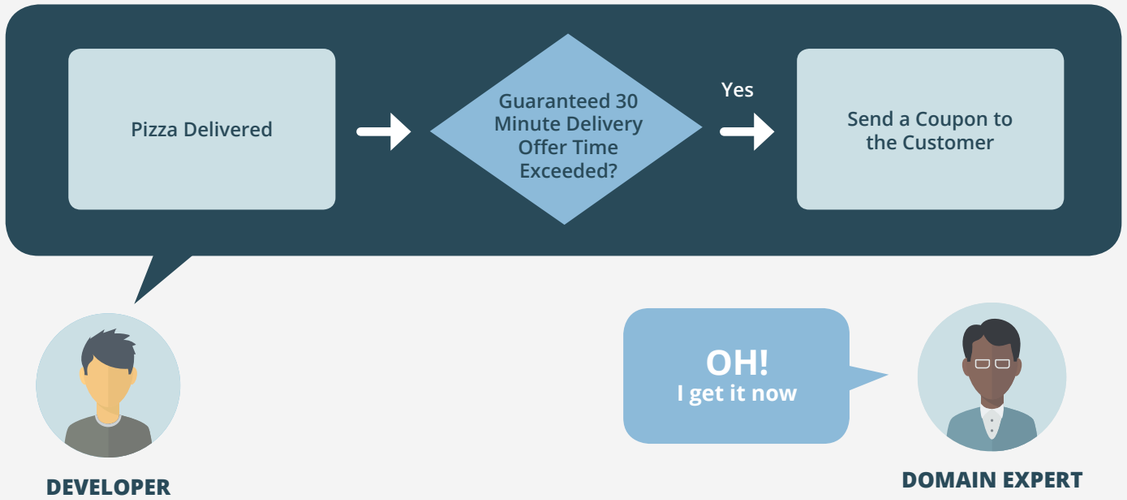
Solution

Decompose large objects structures into smaller objects groupings called aggregates which are based around invariants (business rules). An aggregate is a unit of consistency ensuring transactional boundaries are set at the right level of granularity to ensure a usable application by avoiding blocking at the database level.



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 realized and this is enabled by using common language.



Domain concepts made explicit in the code model

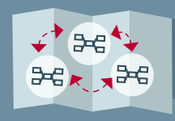
```
public class Guaranteed30MinuteDeliveryOffer
{
        public void After(PizzaDelivered delivery)
        {
                if (delivery.TimeTaken.Exceeded(thirtyMinutes))
                {
                        sendCouponTo(delivery.Customer);
                }
        }
}
```

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

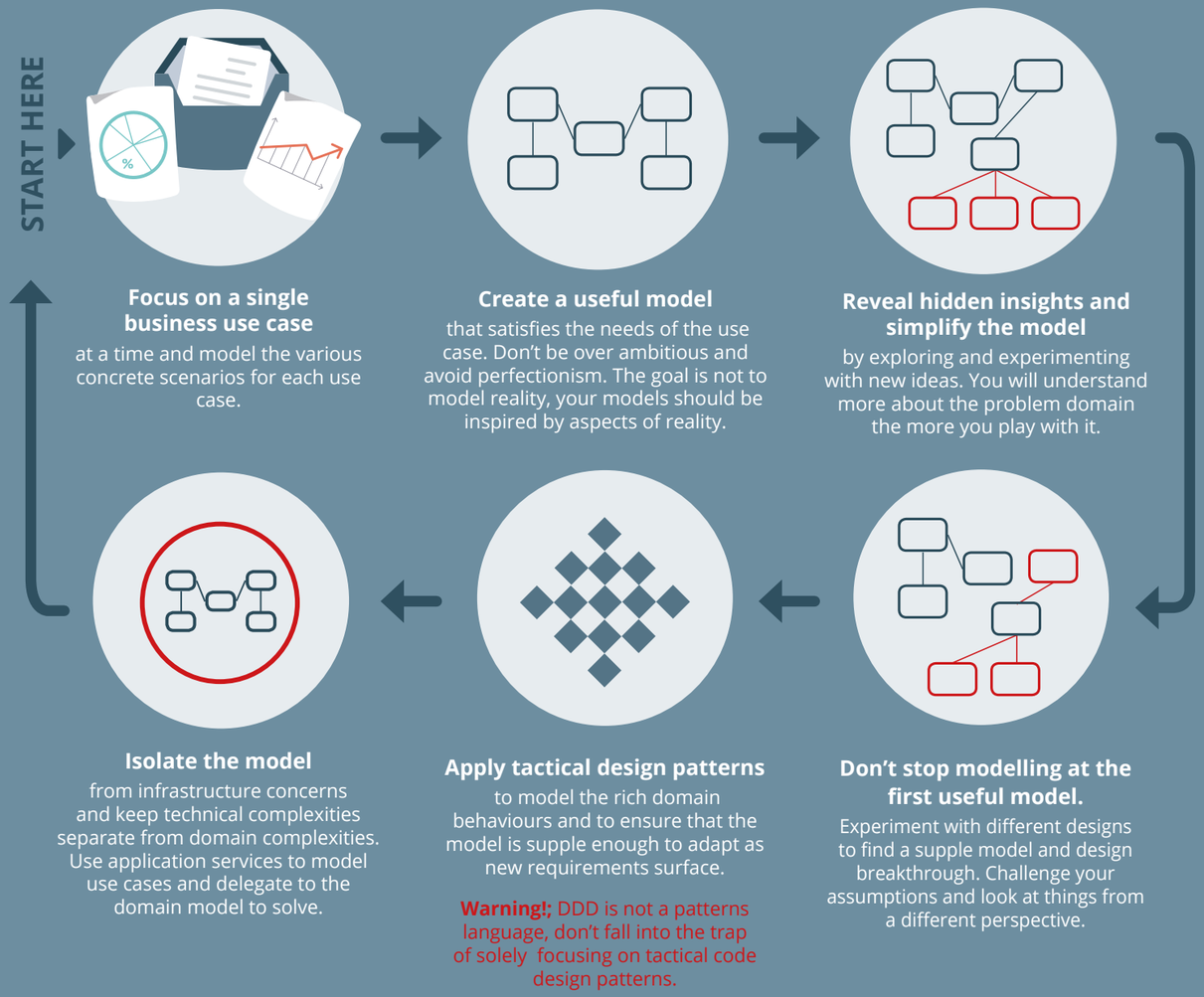
- Martin Fowler, "Refactoring: Improving the Design of Existing Code"

COLLABORATIVE AND CONSTANTLY EVOLVING MODELLING

A big benefit of collaborative modelling is the constant feedback the development team gets from the business experts. This leads to the discovery of important concepts and allows the team to understand what is not important and can be excluded from the model. Breakthroughs in sessions are manifested as simple abstractions that clarify complex domain concepts and lead to a more expressive model.



Before starting to model understand the technical landscape, the relationships with other teams and other models at play.



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.

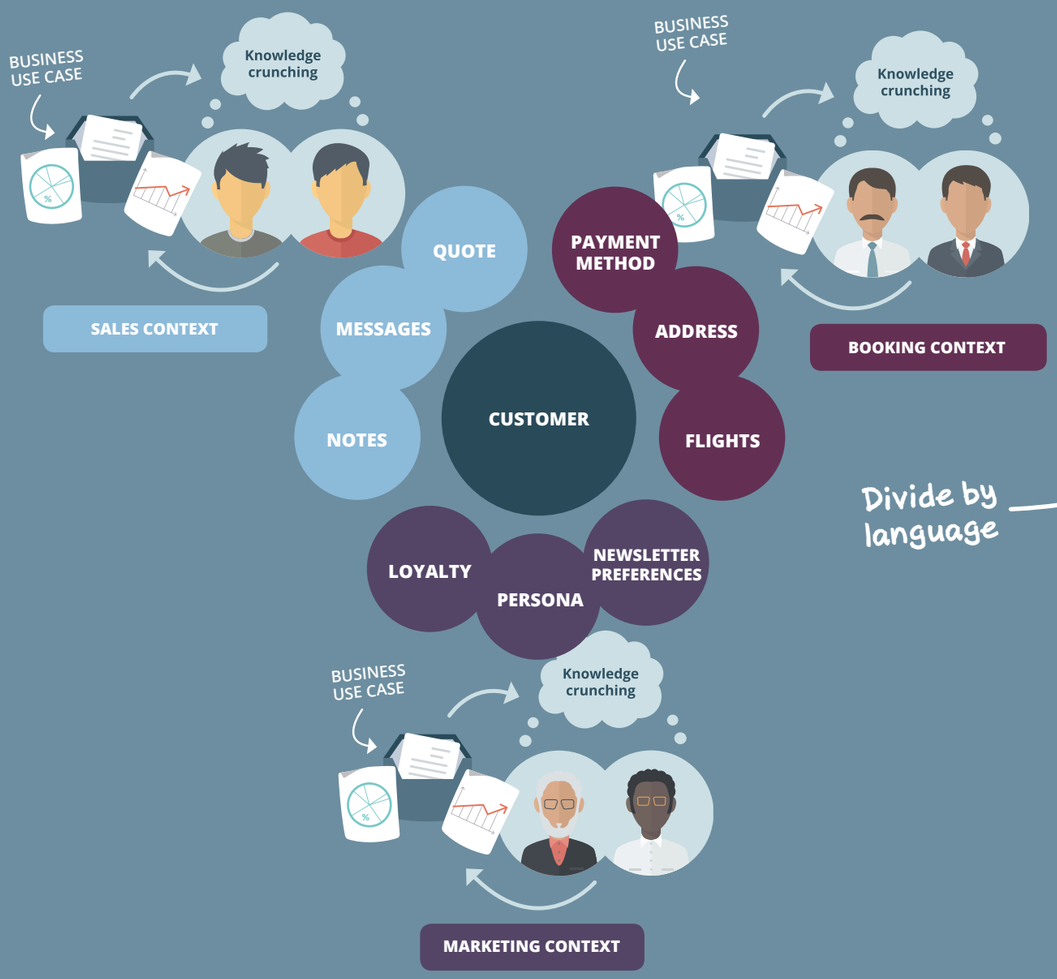
Isolate the model

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.

DIVIDE COMPLEX AND LARGE MODELS INTO SEPARATE BOUNDED CONTEXTS

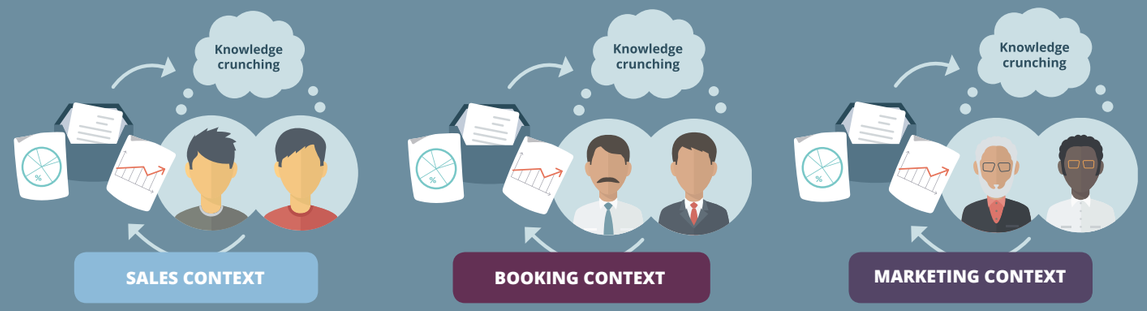
Over time a model will lose integrity and explicitness as it; grows in complexity, multiple teams work on it or language becomes ambiguous. If this is the case, large or complex models should be divided into bounded contexts where a model can be explicit by being understood in a specific context. Software that fails to isolate and insulate a model in a bounded context will often slip into the Ball of Mud pattern.

A Single Large Ambiguous Model

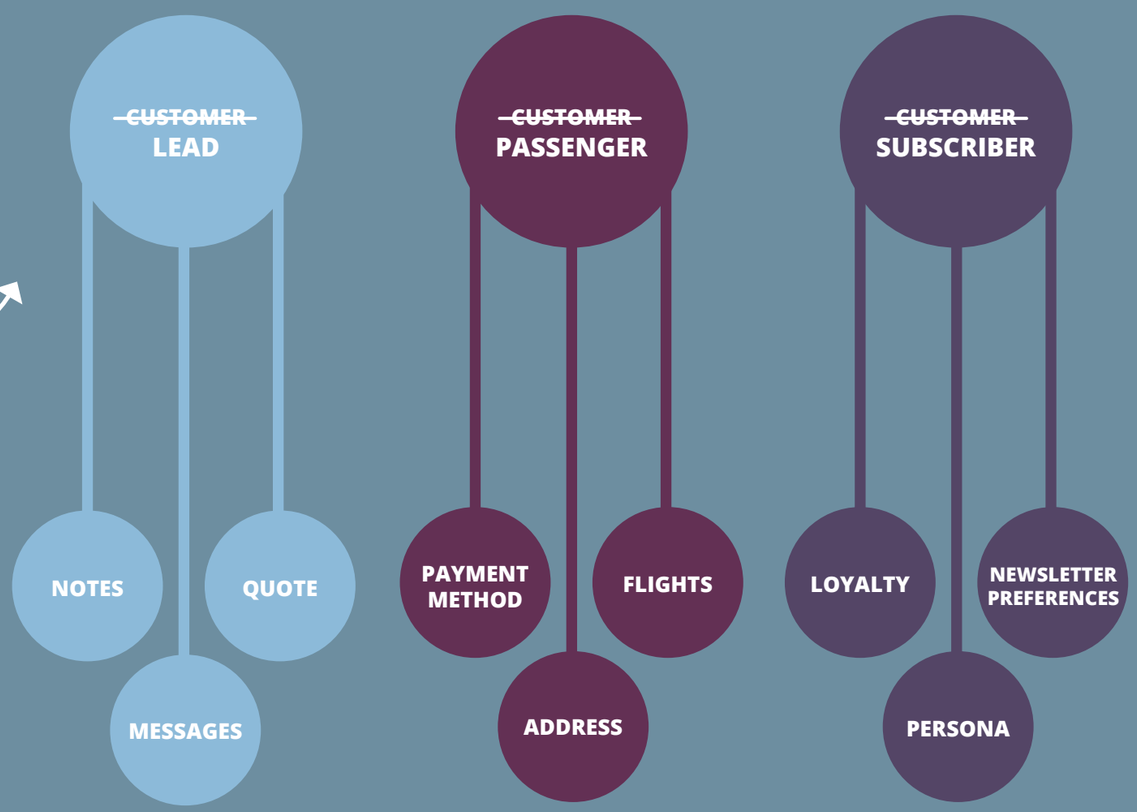


Divide by language

Multiple Smaller Explicit Models

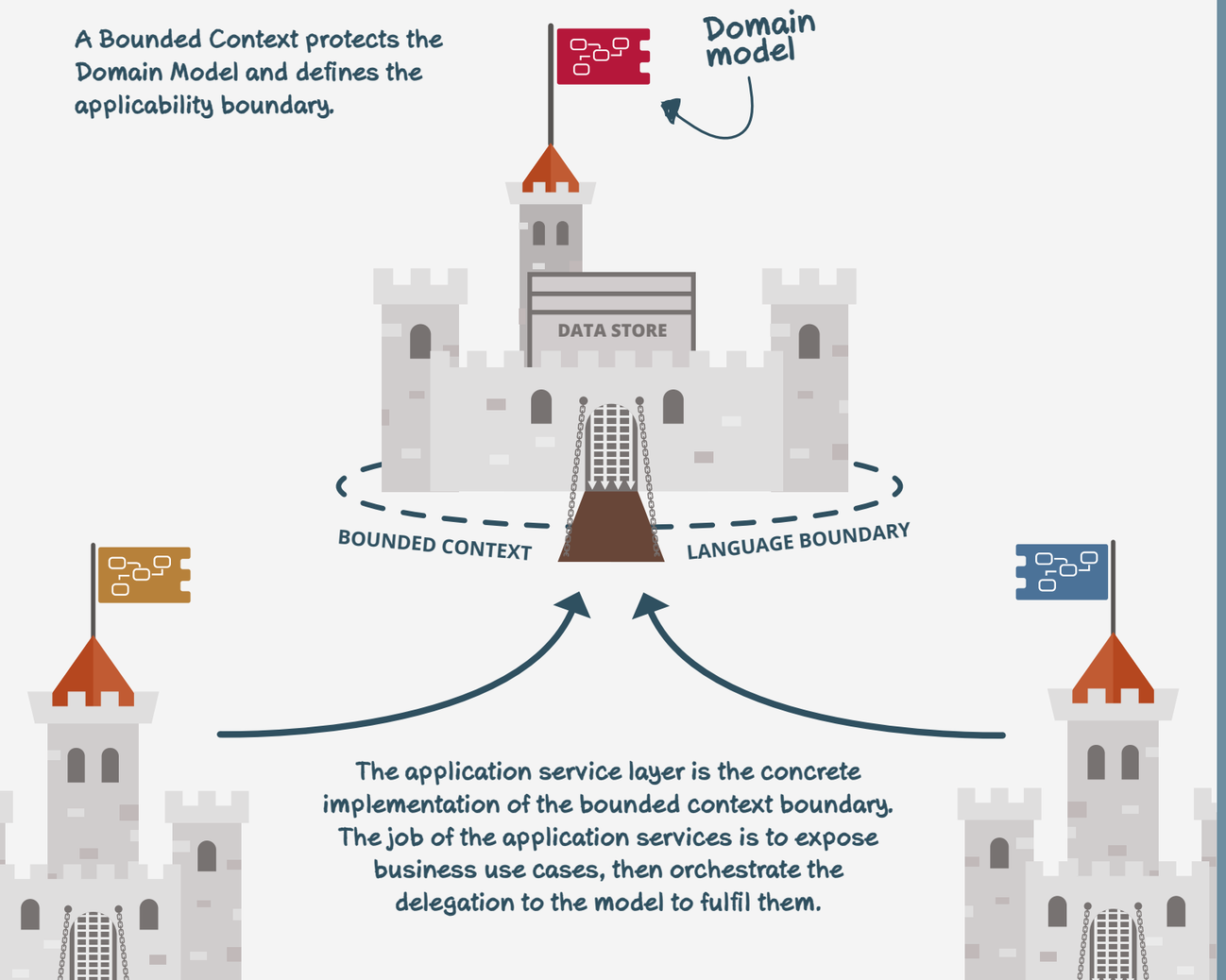


Use Specific Terminology In Each Bounded Context.



IMPLEMENT A BOUNDED CONTEXT TO PROTECT A DOMAIN MODEL

In order to effectively maintain the integrity of a domain model it is important that a bounded context encapsulates the infrastructure, data store, and in some cases the user interface, whilst exposing a set of application services to allow client and other bounded contexts to interact with it. As with a domain model the architectural patterns used to implement bounded contexts should be appropriate to the complexity. If you don't have complex logic in a bounded context and a simple model, use a simple create, read, update, and delete (CRUD) architecture. DDD does not dictate any specific architectural style, it only requires the model to be kept isolated from technical complexities so that it can focus on domain logic concerns.



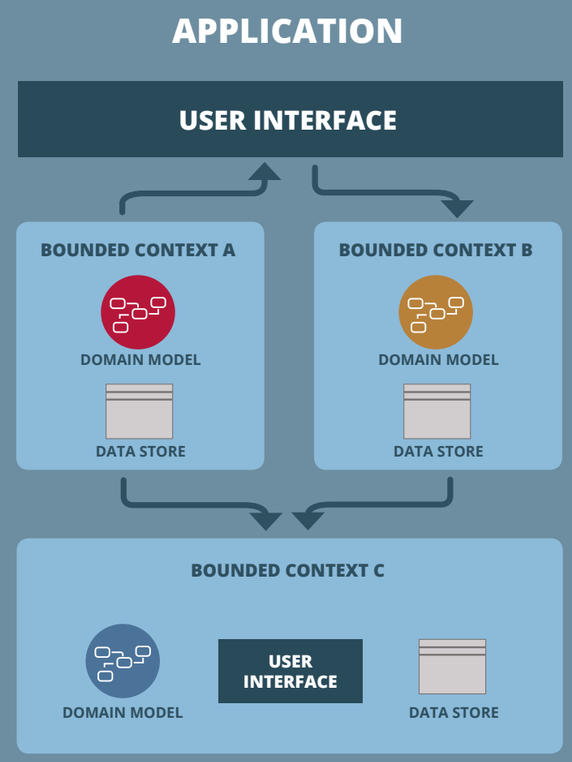
COMPOSE BOUNDED CONTEXTS TO CREATE APPLICATIONS

Applications can be composed of one or more bounded contexts. Where a user interface exists data can be displayed using one of the following integration styles:

1) Authoritative/Composite: The bounded contexts owns the UI and shows only data produced by that BC, or delegates to other bounded contexts directly for UI matters. Alternatively a thin UI delegates to several bounded contexts.

2) Autonomous: The bounded context owns the UI but copies data from other contexts and stores a local copy.

Communication between bounded contexts can be achieved in many ways but try and enforce physical boundaries to enable clean models and to keep bounded contexts autonomous. A great text on integration styles is Enterprise Integration Patterns Book by Gregor Hohpe.



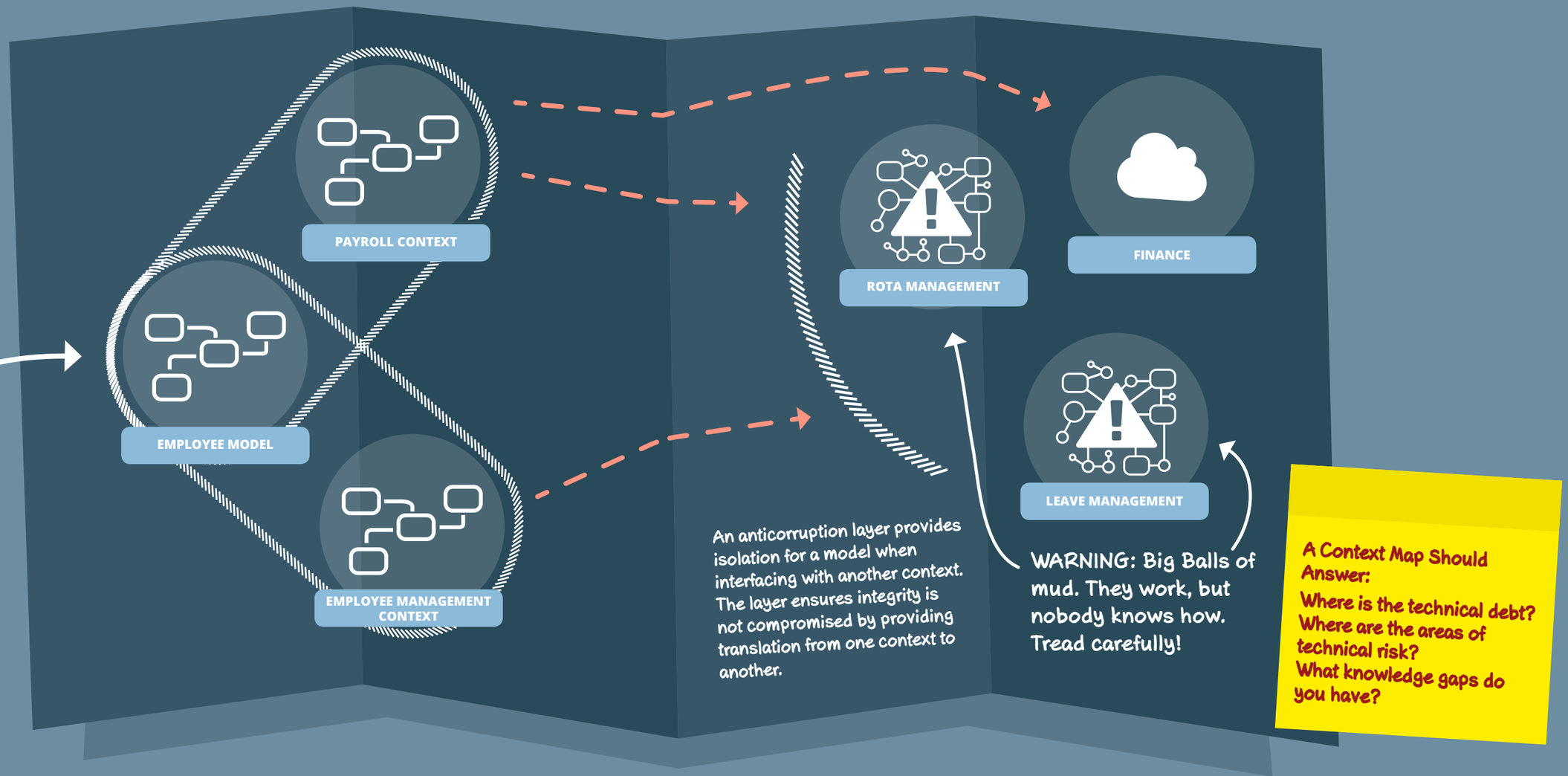
Some bounded contexts may not have a UI. An application's UI maybe a composite of many bounded contexts.

Integration Patterns enable Bounded Contexts to communicate

IDENTIFY AND MAP ALL MODELS AT PLAY, AND HOW THEY RELATE TO EACH OTHER USING A CONTEXT MAP

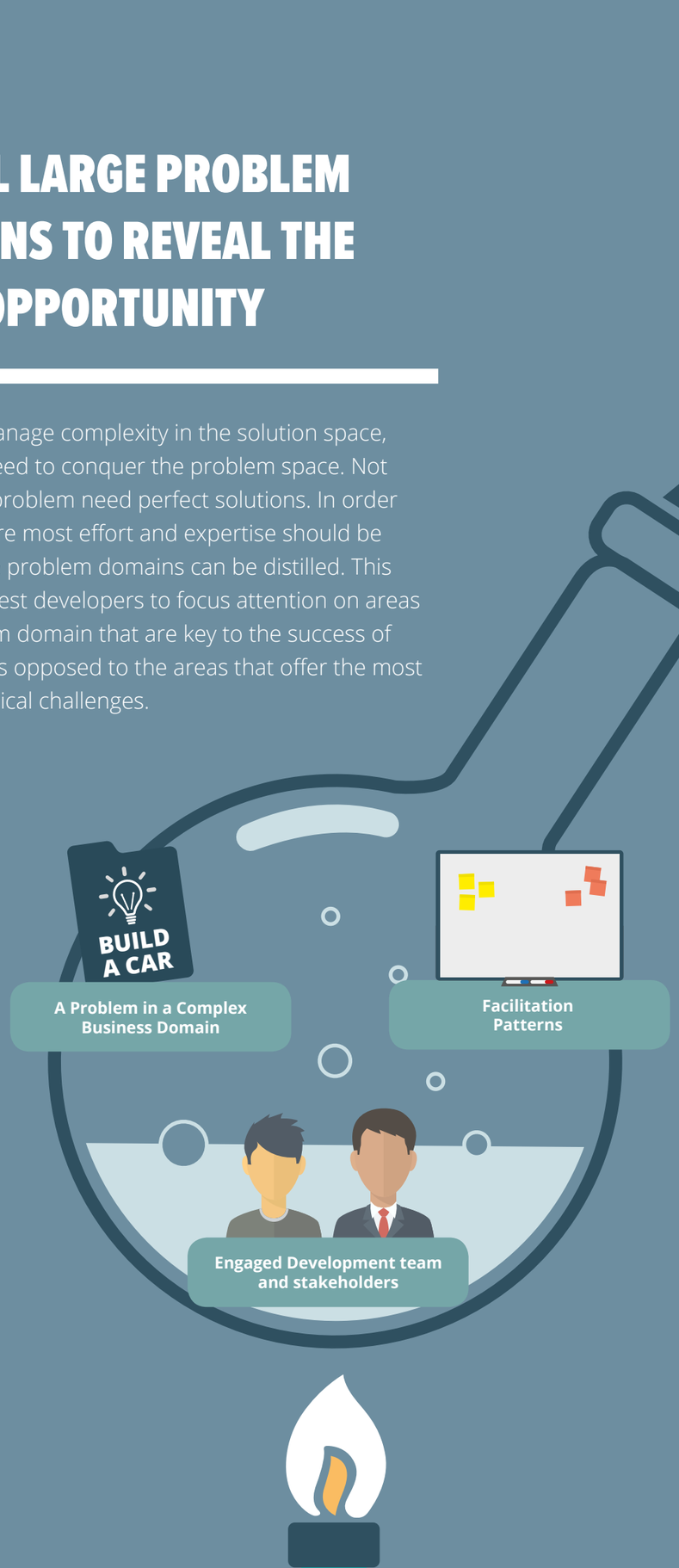
A context map is an important artifact; it's responsibility is to ensure that boundaries between various contexts of the system are designed explicitly and that each team understands the contact points between them. A context map is not a highly detailed document created in some kind of enterprise architecture tool, it is a high-level, hand drawn diagram that communicates a holistic picture of the contexts in play.

A context map should be simple enough to be understood by domain experts and development teams alike. As well as clearly labelling the contexts the teams understand, the diagram should also show areas of the system that are not well understood to reflect the messy and often unintelligible reality of the codebase.



DISTILL LARGE PROBLEM DOMAINS TO REVEAL THE CORE OPPORTUNITY

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.



What makes the system worth writing?

What is the opportunity cost of writing this software? Why has this project been approved over others, what capability or opportunity does this application enable? Developers are expensive, why is this project not being outsourced, what strategic advantage is there in having in house developers work on this project?

Why not buy it off the shelf?

If you can't build it cheaper, faster or better then why build it at all? What reason is driving your company to develop this rather than looking at some off the shelf software. What part of the proposed system will enable the business strategic? Your business has a strategy; software helps to enable that strategy. Understand the benefits that this project will realize, share the goals of the business stakeholders.

Ask powerful questions

What does good look like? What is the success criteria of this product? What will make it a worthwhile endeavor? 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.

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.

Leverage Facilitation Patterns

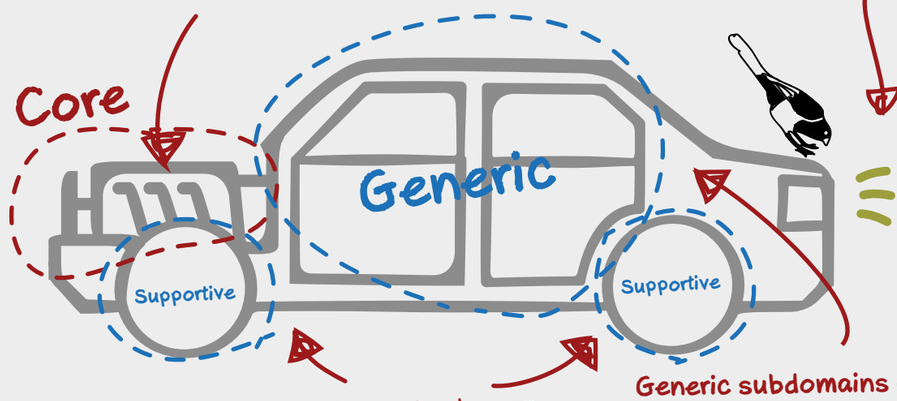
Jeff Pattons' user story mapping, Alberto Brandolini's event storming techniques and Impact Mapping by Gojko Adzic are three great ways to engage stakeholders and reveal the core of the product. (This has been mentioned before, but learn how to facilitate knowledge crunching sessions, it's very, very important!).

What parts of the system will support the core domain?

Not all parts of the application will be of strategic importance, some parts will be in place to support the core domain. Look for areas that need to work but aren't key to the success of the project.

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!

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



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

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

THE SALIENT POINTS OF DOMAIN-DRIVEN DESIGN

To build effective and maintainable software for complex domains you need a dedicated team of software experts working in a iterative development cycle. But as Eric Evans observed in his book Domain-Driven Design, technical practice and expertise will only get you so far. Without a focus on the most important parts of the problem domain, an environment where you can collaborate with domain experts, an obsession with a ubiquitous language and the understanding that concepts need to be understood in context, you may end up with a ball of mud when juggling technical and domain complexity.

FOCUS EFFORT WHERE IT MATTERS

Not all of a system will be well designed and often, it isn't cost effective to strive for this. Instead identify the core domain and the core complexity and focus effort there. The core domain is the reason you are writing the software in the first place. DDD is expensive and time consuming so use it where it matters. Sometimes it's better just to get on and code, rather than looking for complexity where there is none. Not all projects will have suitable complexity that warrants the effort of DDD. If you have an appreciation for the problem space and an empathy for your business you will be in a better position to judge the opportunity cost as you align effort.

DESIGN A MODEL WITHIN A BOUNDED CONTEXT

When creating a model for a large domain it can lose explicitness if there are multiple teams involved, where different language is used or where concepts mean different things in different contexts. Therefore, just as you distill the problem domain to reveal multiple sub domains, you must also decompose the solution space and develop models within explicit boundaries. Context is everything; context and isolation ensure the integrity of your code. It reduces cognitive load and enables multiple teams to work autonomously.

BIND EXPRESSIONS OF THE MODEL USING A UBIQUITOUS LANGUAGE

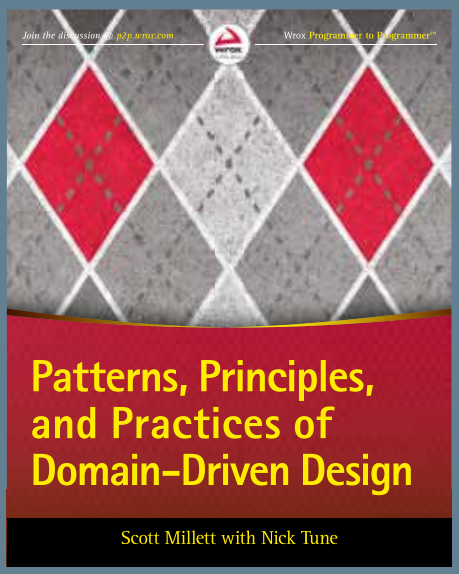
Software projects fail due to poor communication coupled with the overhead of translation between domain and technical terminology. A Ubiquitous Language enables software experts to bind the code model to other expressions of the domain model, such as conversations and diagrams with domain experts, making for more effective communication. Better communication gives you an increased chance to reveal deeper insights in the model. This is why it is vital that the code model is expressed explicitly using the Ubiquitous Language and why it's important to obsess over language. And remember, a language should be specific to a bounded context.

COLLABORATE IN LEARNING AND MODELLING

Don't underestimate the power of collaborative modelling and learning between domain experts and software experts. Knowledge crunching is an ongoing process; collaboration and engagement with the business should not be constrained to the start of a project. Deep insights and breakthroughs only happen after living with the problem through many development iterations. Facilitation patterns to help crunch domain knowledge are extremely important - get good at mining for information and engaging with the business! DDD is the process of learning, refining, experimenting, and exploring in the quest to produce an effective model. It is often said that working software is simply an artifact of learning.

PATTERNS, PRINCIPLES, AND PRACTICES OF DOMAIN-DRIVEN DESIGN

BY **SCOTT MILLETT** & **NICK TUNE**



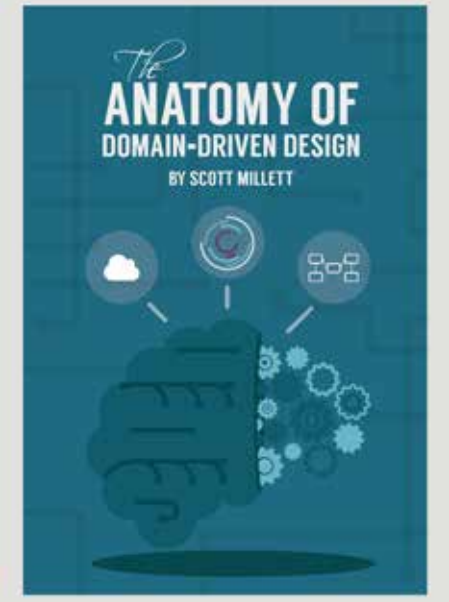
Build solutions for complex business problems more effectively with Domain-Driven Design

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

READ MORE ABOUT THE ANATOMY OF DOMAIN-DRIVEN DESIGN AT:

HTTPS://LEANPUB.COM/ANATOMY-OF-DDD/



The goal of Domain-Driven Design is not to simply produce better software but to enable better business outcomes

This book is a concise, practical and visual guide to the software practice of domain-driven design. While much has been written about the tactical and strategic technical patterns of DDD many of the practices and principles required for empathic development have had little attention.

DDD is deceptively simple. Fundamentally DDD is about minimising the cost of translation from problem domain to software solution. It is this area that developers continue to struggle, and it is this area that is the focus of this book.

This book offers:

- ✔ A plain English, highly-visual overview, introducing you to all aspects of DDD
- ✔ Facilitation patterns that empower you to explore business domains and discover the all-important core domain(s)
- ✔ Collaboration techniques that help you to become an expert in your domain's language & work effectively with domain experts
- ✔ Knowledge crunching and modelling techniques that teach you how to make sense of your domain and accurately model it
- ✔ Organisational design patterns that show you how to create autonomous teams by aligning them with the problem domain
- ✔ Leadership guidance, advising on how to get your whole team motivated and involved in DDD