DOMAIN DRIVEN DESIGN

The First 15 Years

Essays from the DDD Community

Sponsored by
Domain-Driven Design Europe

Domain-Driven Design: The First 15 Years

Essays from the DDD Community

The DDD Community

This book is for sale at http://leanpub.com/ddd_first_15_years

This version was published on 2024-01-11

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help The DDD Community by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I'm reading "Domain-Driven Design: The First 15 Years - Essays from the DDD Community" https://dddeurope.com/15years #DDDEU #DDDesign

The suggested hashtag for this book is #DDD15years.

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

#DDD15years

Dedicated to Eric Evans

Contents

Foreword

It's rare for a software book to last fifteen years. By the time a new book gets out of beta, it's already at risk of being obsolete. Eric Evans' *"Domain-Driven Design — Tackling Complexity in the Heart of Software"* (Addison-Wesley) is, over 15 years after its publication, sparking a renewed interest in software design. Books and blogs have expanded on the ideas proposed by Eric; people have created new methods to apply the principles; there are workshops and online courses; conferences in Europe, Asia, and North America; and dozens of meetups all over the world.

And where traditionally the DDD community used to be exclusively populated by programmers and architects, we're now seeing growing attention from different disciplines in software design. Analysts seem to be leading the way — a natural fit, as modelling has always been a fundamental part of analysis. But now testers and product designers are discovering the value of Domain-Driven Design. They too deal in models, and are attracted to the principles and methods of building and collaborating on models, of sharing a Ubiquitous Language, and of finding better context boundaries for managing the growing complexity of software.

What makes the success of DDD even more surprising is that Eric's book has a reputation of being theoretical, academic, or philosophical; terms used by programmers when they simply mean "hard". It has to be: it was extraordinarily ambitious to write a book that deals with complexity from the smallest domain object to large scale structures. DDD is hard because software design *is* hard. Once you get used to the density of knowledge in Eric's book, you'll find it is in fact highly practical. The ideas originated in the real world, in highly complex environments, and are hardened by years of deep thinking, discussions, and experiments.

DDD is not "done". Last Summer, during a dinner in Paris, Eric spoke about how he'd love to see more publications about Domain-Driven Design. There's no lack of interesting new ideas in this community, but they are dispersed over talks, blogs, Twitter, and mailing lists. So to celebrate DDD's 15th anniversary, we couldn't think of a better idea than to make this book. We hope it will inspire others to develop new ideas and to write.

When we approached authors for this book, we deliberately imposed as few constraints as possible. The result is a highly diverse collection of essays. Some authors have been there since the start of Domain-Driven Design, others are fairly new to this community. Some entries were written specifically for this book, others are adaptations from older work — Martin Fowler's entry, not coincidentally, even predates DDD. The topics are eclectic as well, from philosophical meanderings to deep technical discussions, from tried and true methods to experimental ideas, from critical analysis to DDD love letters.

Eric, we present this book to you during the Domain-Driven Design Europe 2019 conference, as a token of our appreciation for your generosity in sharing your ideas. Here's to the next 15 years!

Mathias Verraes

Founder Domain-Driven Design Europe

Acknowledgments

I'd like to thank Anneke Schoonjans, Céline Dursin, Paul Rayner, and Indu Alagarsamy for their help with the logistics and content of this book. The wonderful cover design is by Nick Liefhebber. And most importantly, I'm incredibly thankful for the more than 20 authors who contributed to this book. You are all people who have in one way or another inspired me, and countless others.

About DDD Europe

Domain-Driven Design Europe is the world's leading DDD conference. Our mission is to expose the community to new ideas and influences from inside and outside the DDD community, and to spread DDD throughout the software industry. Please visit the DDD Europe site[1] to learn more, and we hope to welcome you as a participant soon.

Corrections

Corrections or comments can be submitted via GitHub[2] or to the authors directly.

[1] https://dddeurope.com
[2] https://github.com/mathiasverraes/15yearsddd

Distilling DDD Into First Principles — Scott Millett

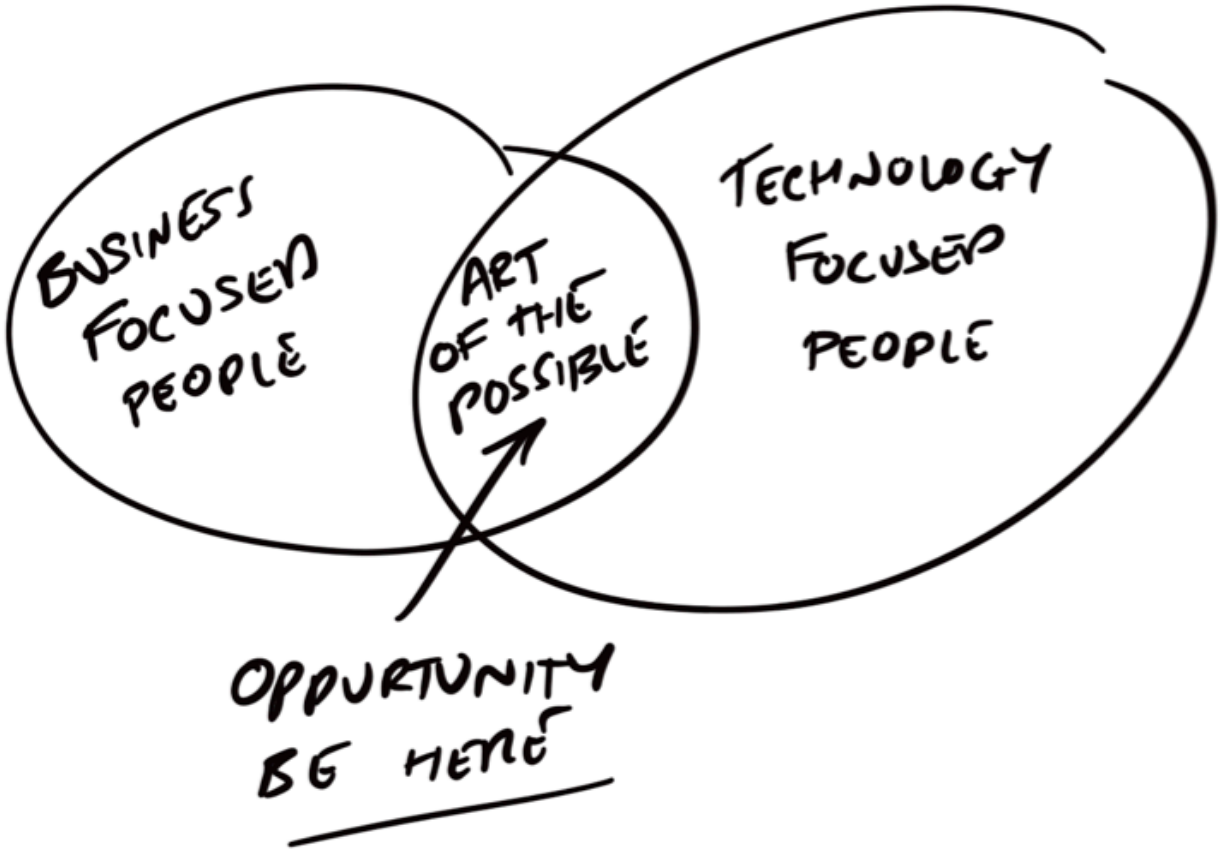
Parts of this essay first appeared in the book Patterns, Principles, and Practices of Domain-Driven Design (Wrox 2015) by Scott Millett and Nick Tune.

If I could offer you one piece of advice (apart from always wear sunscreen), it would be to do your utmost to move beyond simply understanding your domain and get to a position where you are in sync with your business vision, goals, strategy and constraints. You should share the same worries as your business counterparts: are we going to hit budget? how do we successfully launch our proposition in a new territory? how do we remove the bottleneck in fulfillment? If you are able to move to a more fundamental level of understanding for your business then you should be in a position that not only gives you a deeper level of understanding for your domain, enabling you to produce effective solutions, but will also allow you to proactively identify product opportunities that offer true business value.

It should come as no surprise to you that software is eating the world and that IT is now a crucial capability in any organisation. However most technical teams only seem to have a shallow understanding of the domain they work within and in my experience are more focused on tech for tech's sake over the overall strategy and needs of a business. Advances in technology have afforded a huge amount of opportunity for business development and evolution, however the majority of businesses are yet to catch up on these opportunities. This is not due to a lack of technical savviness from business colleagues but more from a lack of alignment from technical people, those that understand the art of the possible, which prevents them from seeing opportunities.

The line between the non-technical and technical people within modern businesses is blurring. I am seeing more progressive companies that have blended roles where technical people have a seat at the table. They have a seat because they have earnt it through leveraging the art of the possible, technical opportunities, to remove a constraint on the production of business value. They are aligned to the real needs of the business and they have the autonomy to deliver solutions to meet those needs.

This essay is about my story and how DDD helped me learn more about solving problems and helping people to refocus efforts on how solutions are derived, how people communicate and collaborate, and how technical people can deliver solutions and identify opportunities that exceed expectations of business colleagues.



Be the art of the possible

Disclaimer

> "I don't have talent, so I just get up earlier." —Henry Rollins

This essay is all from my perspective and therefore it is heavily influenced by my own experience. I am not a consultant nor have I worked as a contractor so my experience is limited to the domains of e-commerce in one flavour or another. However in my opinion these domains have been sufficiently complex and thus have benefited from applying the practices of Domain-Driven Design. I am not a jedi master in the art of Domain-Driven Design, I did not attend the Eric Evans finishing school (does that exist? If so please send me details) so I am unable to offer any words of wisdom less the ones that I have come to regard as lessons I have learnt.

Over my career I have had the great fortune of working in a industry full of the most generous of people from all over the world. The community of software is very good at offering opinions, advice and experience - all of which needs to be understood in the context that it is offered and of course moulded to the context that it will be applied in.

So while I can't guarantee that my words of wisdom will fix all your problems (or any), I do hope you find something in the lessons that I have learnt over the years that perhaps can act as a catalyst to helping you on your DDD journey. However do remember, its all about context, and this is mine.

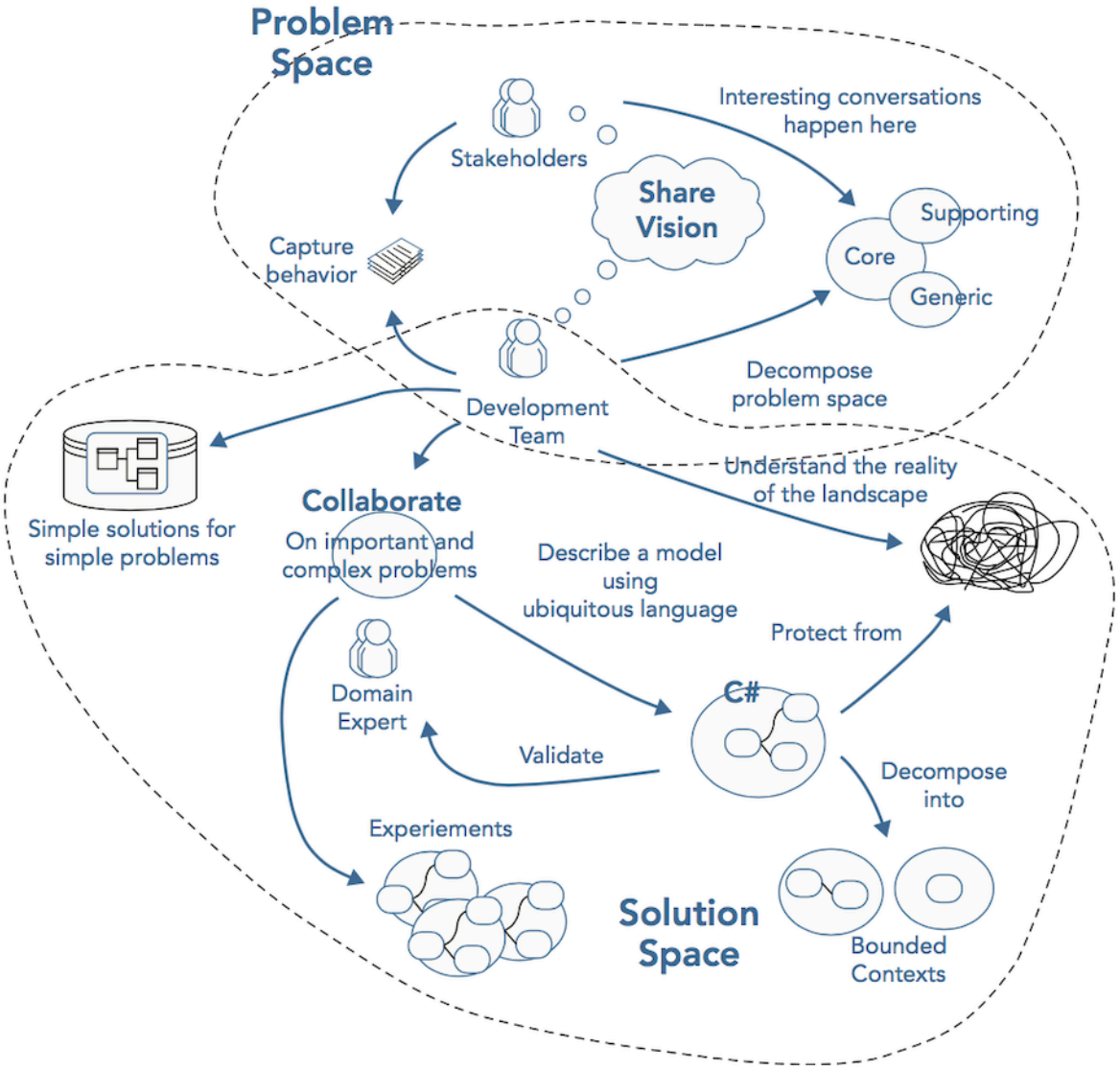
The Fundamental Concepts Of DDD

> Domain-driven design is both a way of thinking and a set of priorities, aimed at accelerating software projects that have to deal with complicated domains. Eric Evans, Domain-Driven Design (2003)

Before I talk about my takeaways from DDD and my view on the first principles I want to quickly recap on the fundamental concepts of Domain-Driven Design as it is often misunderstood when in my opinion it is deceptively simple. Of course the devil is in the detail.

DDD in a nutshell:

- Distill a large problem domain into smaller sub domains.
- Identify the core sub domains to reveal what is Important. The core domains are those of greater value to the business which require more focus, effort and time.
- Collaborate with experts to discover an analysis model that will provide solutions to solve problems or reveal opportunities particularly in the core domain.
- Use the same ubiquitous language to bind the analysis model to the code model. Use tactical patterns to separate technical code from domain code to prevent accidental complexity.
- Split the model (if necessary) into smaller models where there is ambiguity in language or the model is too large for a single team. Enclose the model within a boundary to protect the models integrity. When working with multiple models it's important that they are understood in context.
- Keep a context map to understand the relationships, social and technical, of all models in play.



Overview of Domain-Driven Design

Distill The Problem Domain To Reveal What Is Important

Development teams and domain experts use analysis patterns and knowledge crunching to distill large problem domains into more manageable sub domains. This distillation reveals the core sub domain(s)—the reason the software is being written. The core domain is the driving force behind the product under development. For example in a airline pricing system the algorithm could be the key to the software being successful or not. The system would of course need identity and access control management but this would only be to support the core domain. In a different domain, say government documentation control the security and access may be the core domain

and management of content only supporting. The point is that DDD emphasizes the need to focus effort and talent on the core sub domain(s) as this is the area that holds the most value and is key to the success of the software.

Create Models To Solve Problems

With an understanding of where to focus effort, the technical team along with domain experts can begin to derive a solution represented as an analysis model. This is typically done in a collaborative manner occuring around whiteboards, working through concrete scenarios with business experts and generally brainstorming together. This process is the catalyst to conversation, deep insight, and a shared understanding of the domain for all participants. It is the quest to discover and agree on a shared understanding of the problem domain to produce a model, using a shared, ubiquitous Language, that can fulfill business use cases, remove constraints or open up opportunities.

When a useful analysis model is discovered a code model can follow. I hasten to add that this process is not as linear as I am explaining here and often a code model is used during model exploration to prototype ideas in code to understand feasibility. The code model is to bound to the code analysis model via the use of a shared, or as DDD refers to it a Ubiquitous Language. The Ubiquitous Language ensures that both models stay in sync and are useful during evolution. Insights gained in either model are shared and knowledge is increased, leading to better problem solving and clearer communication between the business and development team.

Tactical patterns are used to keep the code model supple and to isolate domain from infrastructure code thus avoiding the accidental complexity of merging technical and business concerns.

Split Large Models To Prevent Ambiguity And Corruption

Large models can be split into smaller models and defined within separate bounded contexts to reduce complexity where ambiguity in terminology exists or where multiple teams need to work in parallel. The bounded context defines the applicability of the model and ensures that its integrity is retained. The bounded contexts form a protective boundary around models that helps to prevent software from evolving into a big ball of mud. Context boundaries aren't limited to just language or team set up. They can be influenced by ambiguity in terminology and concepts of the domain, alignment to subdomains and business capabilities, team organization for autonomy and physical location, legacy code base, third party integration and a host of other factors.

Why Teams Need To Realign With DDD's First Principles

As you will have noticed, the majority of effort when applying the practices of Domain-Driven Design lie outside of the technical realm. Many people's first introduction with Domain-Driven Design is an exposure to the tactical design patterns as well as techniques such as event sourcing

or CQRS. They never experience or learn to appreciate the true power of DDD because they aren't aware or don't focus on the non technical aspect of software creation. It is the deep understanding of the problem space and the relentless focus on the core domain that allow effective and collaborative model-driven designs to lead to a viable solution. Only then do we need to leverage tactical patterns to organise code in such a manner as to reduce accidental complexity.

Why Domain-Driven Solutions Often Fail To Deliver

The reason that solutions fail to deliver, and the reason we need to focus on first principles, is not because of a lack of programming ability or technical expertise, but rather because of a lack of understanding, communication, and business knowledge. I am not saying technical people are lazy just that output doesn't always translate to outcome. This lack of understanding stems from how developers capture knowledge of the problem domain they work in. Put another way, if developers and customers cannot effectively communicate, aren't aligned on the same overarching goals then even with the most accomplished programmers in the world, you ultimately cannot produce meaningful outcomes.

Having the right soft skills to avoid becoming entrenched or attached to an early version of a model is also important. For example having a technically supple model is great, but useless if you're not able to realise it doesn't work as your initial understanding was proved to be incorrect.

Striving For Tactical Pattern Perfection

Teams concerned only with writing code focus on the tactical patterns of DDD. They treat the building block patterns as a bible rather than a guide, with no understanding of when it's okay to break the rules. They waste effort adhering to the rules of the patterns. This energy is better spent on understanding why it needs to be written in the first place. DDD is about discovering what you need to write, why you need to write it, and how much effort you should use. The tactical patterns of DDD are the elements that have evolved the most since Eric's book was written, with the strategic side of DDD remaining faithful to Eric Evan's original text, albeit there has been great progress on the techniques of how to distill problem spaces which I will discuss later. How development teams create domain models is not nearly as important as understanding what models to write in the first place and how to develop them in a bounded context. Understanding the what and the why of problem solving is a more important process to get correct than how you are going to implement it in code.

Over Valuing Sample Applications

One of the most often-asked questions on software development forums is "Can I see a sample application?" There are probably many good solutions that show the result of a product developed under a DDD process, but much of the benefit of DDD is not revealed when you only examine the code artifacts. DDD is performed on whiteboards, over coffee, and in the corridors with business experts; it manifests itself when a handful of small refactorings suddenly reveal a hidden domain

concept that provides the key to deeper insight. A sample application does not reveal the many conversations and collaborations between domain experts and the development team nor does it reveal the "Aha!" moments when deeper understanding of the problem domain is discovered.

The code artifact is the product of months and months of hard work, but it only represents the last iteration. The code itself would have been through a number of guises before it reached what it resembles today. Over time, the code will continue to evolve to support the changing business needs; a model that is useful today may look vastly different to the model used in future iterations of the product.

If you were to view a solution that had been built following a DDD approach hoping to emulate the philosophy, a lot of the principles and practices would not be experienced, and too much emphasis would be placed on the building blocks of the code. Indeed, if you were not familiar with the domain, you would not find the underlying domain model very expressive at all.

Missing The Real Value Of DDD

A team focusing too much on the tactical patterns is missing the point of DDD. The true value of DDD lies in the creation of a shared language, specific to a context that enables developers and domain experts to collaborate on solutions effectively. Code is a by-product of this collaboration. The removal of ambiguity in conversations and effortless communication is the goal. These foundations must be in place before any coding takes place to give teams the best chance of solving problems. Problems are solved not only in code but through collaboration, communication, and exploration with domain experts. Developers should not be judged on how quickly they can churn out code; they must be judged on how they solve problems with or without code.

Going Back To First Principles

The first principles emphasise that focus for technical teams should be more aligned with the domain and driving all decisions from that position rather than only on technical concerns. After all with DDD the clue is in the name, the domain is the business, drive all design decisions based on the specifiction "will this help me achieve the business goal?". If you don't have a solid understanding of the goal then it is unlikely you will make good decisions. All decisions need to be taken in the wider context of your business and how what you do enables value to be produced.

In the remainder of this essay I will present to you each of these five first principles that I have distilled down from the non-technical aspects of DDD and learned to focus on during my career to allow me to really use the power of DDD.

My first principles are:

1. Gain agreement on the problem
    - Focusing on the motivation behind the need for a solution and understanding the problem within the wider context of the business.

    - Contributing to delivering real business value by empathising with your business colleagues regarding the opportunity you are enabling or the constraint you are removing.
2. Collaborate towards a solution
    - Moving past requirements gathering through collaborating on the vision and direction of a solution and the impacts that will resolve the problem.
    - Gaining alignment by investing time and energy at the core problem sub domain.
3. Ensure the solution solves the core problem
    - Focusing on outcomes over output to prevent over complex code and unnecessary investment.
    - Avoiding becoming attached to a solution by keeping things simple and being comfortable that you never know as much as you think you do.
4. Optimize the overall system
    - Having shared accountability for the entire system/process over siloed team optimisation.
    - Aligning on the big picture goal and understanding how you contribute.
5. Be a positive influence on the team
    - Being respectful, having patience and showing passion.
    - Displaying humility and empathy with others whilst reaching solutions as a team.

I would hasten to add that however well you manage the non technical side, it goes without saying you need to have a proficient technical ability. However I have often witnessed that a good understanding of the goal of the core domain and the problem space itself enables a simpler solution to be found that requires a simpler technical implementation.

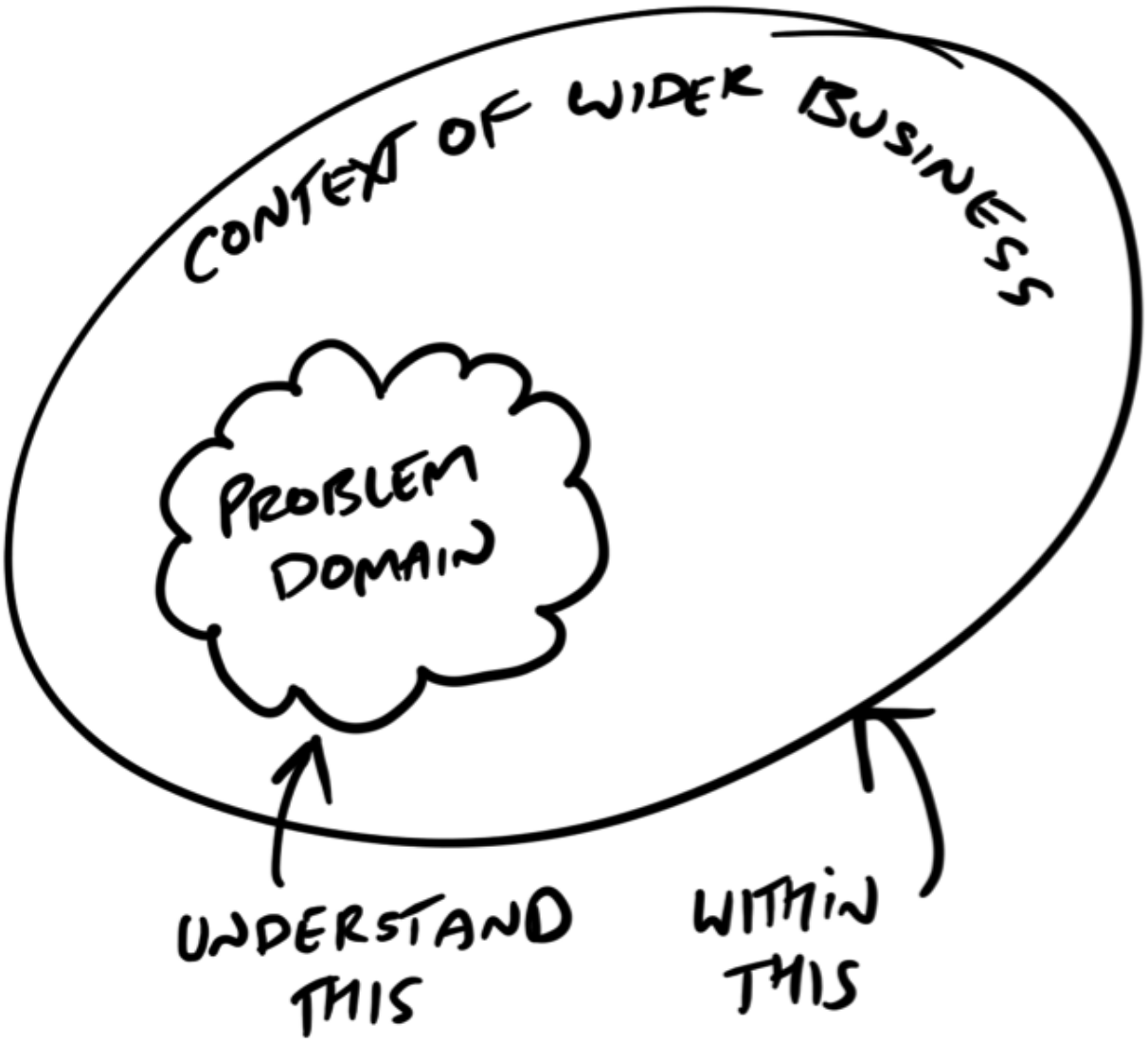
Principle 1: Gain Agreement On The Problem

> "If you do not deal directly with the core problem, don't expect significant improvement."
> —Eli Goldratt

To know where to focus effort you first need to understand the motivation for solving a problem. You need to have a solid understanding of the business big picture so that you are able to understand a problem in context. What problems, constraints, opportunities exist that have led up to this point and proved that this problem deserves a solution?

Why you ask? The reason is to ensure you truly understand what you are being tasked with to solve. As without this fundamental information it is impossible to be certain that you will be able to produce a solution with significant and beneficial outcomes. I will let you in on a secret - your business counterparts don't have all the answers. They are experts in their respective domains, as you are in your technical domain. But they are not system or process design experts. They are hypothesising on what they should do. Therefore instead of taking things at face value, collaborate and empathise so you can see the thought processes of your business counterparts and understand how they arrived at their conclusions. This activity often leads to a technical team being able to

add real value by looking at simpler alternative solutions or problems further up the supply/process chain that may have been missed. Techniques such as impact mapping are great for highlighting this.



Understand the problem domain in contetx of the wider business

It is always worth asking why rather than just accepting that someone else has ensured that this is the correct solution to the business need further up the chain and not simply a local optimisation or a wish list item. How will building an application make a difference to the business? How does it fit within the strategy of the company? Why are we looking at technical solution? Does part of the software give the business a competitive edge? Often the result of systematically asking "why" will reveal a painfully simple solution or a complete change in direction. Clearly, don't ask the question as a child might; be creative with how you search for the truth, but pull on the string of a problem

until you reach the root of the matter.

Of course I am not expecting you to be a proxy CEO and change the direction of a company. But you ought to at least question and understand people's motivation for problem solving at a deeper level. This will enable you, and your team, to buy in and be fully committed to an idea.

Be Clear On How You Deliver Business Value

> "People don't want to buy a quarter-inch drill, they want a quarter-inch hole." Theodore Levitt

A software developer is primarily a problem solver. Their job is to remove blockers that prevent the business producing value rather than producing code. Code should be viewed as the by product of finding a solution to a business problem, meaning that, on occasion, developers can actually solve problems without having a technical solution at all. Focus should be squarely on contributing to business outcomes over software output. It's not about writing elegant code, it's about getting results that benefit the business and help achieve its goal.

The IT department is part of the business, you should strive beyond creating software based on what you understand from the business to creating solutions that contribute to the overall business goal. This is a small but important distinction. Its similar to the paradoxical saying "I'm stuck in traffic" - you are not stuck in traffic, you are traffic. Software developers ARE the business, they just happen to be domain experts in software design just as accountants are domain experts in finance. You should not have a them-and-us mentality.

As I have already mentioned, IT is far more strategic in most businesses than ever before. The advances in technology have disrupted many markets, think of Kodak and Blockbuster, giants businesses that have vanished. Delivering true value stems from acting from a domain perspective and identifying opportunities through leveraging your technical domain knowledge to help your business counterparts understand the art of the possible. Look for the money, how can I help shift the KPI, how can I contribute to removing a bottleneck to business throughput? This is how you win, this is how you deliver real value.

Techniques For Revealing Where Business Value Lies

There are a plethora of techniques to extract where you should focus to enable business value. Such as

- The Business Model Canvas
- The Lean Enterprise Innovation Portfolio
- Wardley Mapping
- Customer Journey Mapping
- Domain Storytelling

I will explain the four practices that I have found particularly useful and powerful to me.

Event Storming

Event Storming is a workshop activity that is designed to quickly build an understanding of a problem domain in a fun and engaging way for the business and development teams. Groups of domain experts, the ones with the answers, and development team members, the ones with the questions, work together to build a shared understanding of the problem domain.

Knowledge crunching occurs in an open environment that has plenty of space for visual modeling, be that lots of whiteboards or an endless roll of brown paper. The problem domain is explored by starting with a domain event; i.e., events that occur within the problem domain that the business cares about. A Post-it note representing the domain event is added to the drawing surface and then attention is given to the trigger of that event. An event could be caused by a user action that is captured and added to the surface as a command. An external system or another event could be the originator of the event; these are also added to the canvas. This activity continues until there are no more questions. The team can then start to build a model around the decision points that are made about events and when they, in turn, produce new events.

Event storming is an extremely useful activity for cultivating a UL as each event and command is explicitly named, this goes a long way to producing a shared understanding between the developers and business experts. The biggest benet however is that it's fun, engaging, and can be done quickly. Alberto Brandolini created this activity and more information can be found at eventstorming.com[3].

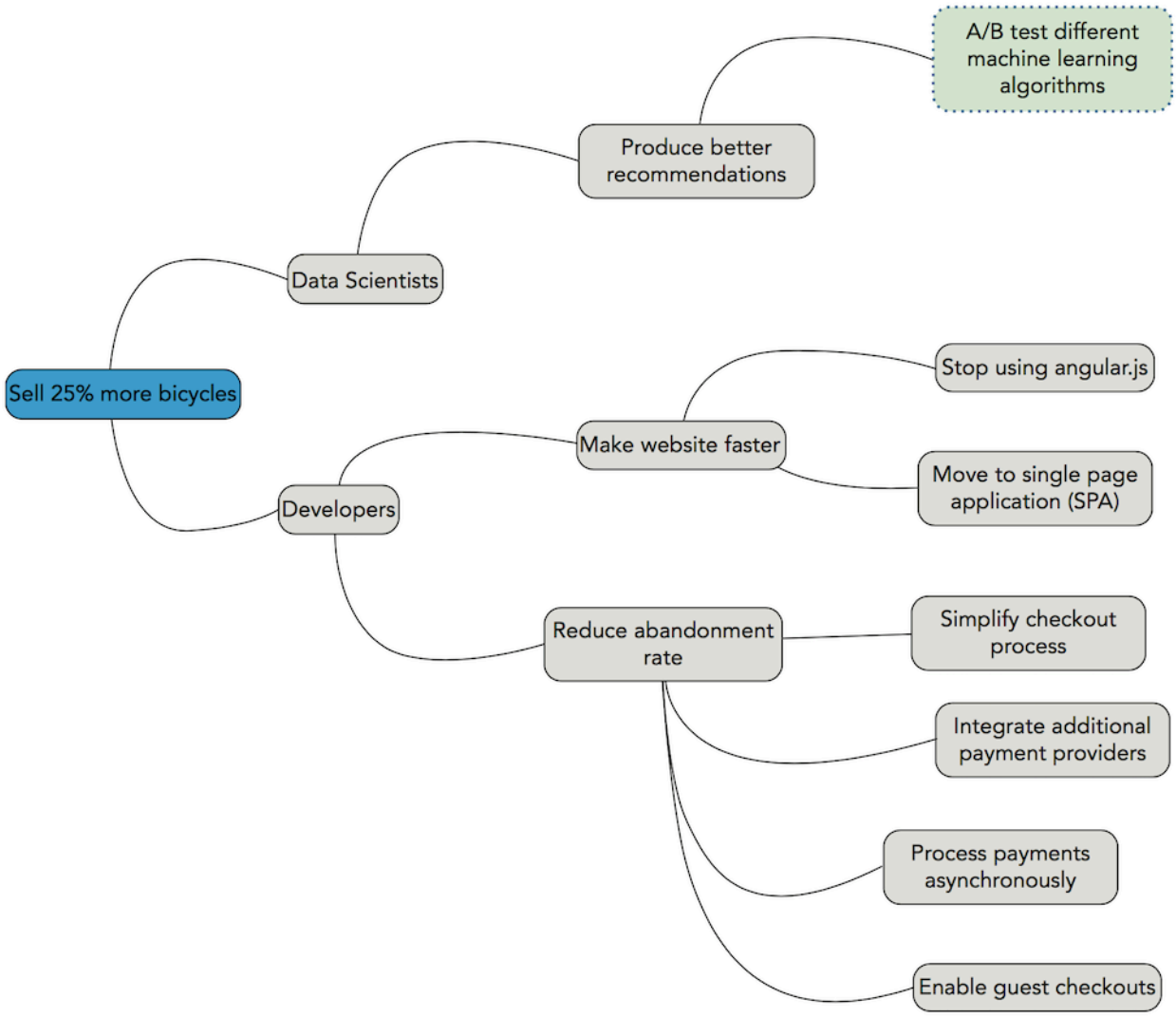
Impact Mapping

A technique for better understanding how you can influence business outcomes is impact mapping. With impact mapping, you go beyond a traditional requirements document and instead you try to work out what impacts the business is trying to make. Do they want to increase sales? Is their goal to increase market share? Do they want to enter a new market? Maybe they want to increase engagement to create more loyal customers who have a higher lifetime value.

Once you understand the impact the business is trying to make you can play a more effective role in helping them to achieve it. Significantly for DDD, you will be able to ask better questions during knowledge-crunching sessions since you know what the business wants to achieve. Surprisingly, impact mapping is a very informal technique. You simply create mind-map-like diagrams that accentuate key business information. You work with the business so that, like knowledge crunching, it is a collaborative exercise that helps to build up a shared vision for the product.

An impact map, rather obviously, starts with the impact. For example "sell 25% more bicycles". Directly connected to the impact are the actors—the people who can contribute to making the desired impact. That would be developers and data scientists. Child nodes of the actors are the ways in which the actors can help. One way the developers can help to create the business impact is to improve the performance of the website so that people are more likely to make a purchase. Finally, the last level of the hierarchy shows the actual tasks that can be carried out. You can see in the image that one way the developers may be able to make the website faster is to remove slow frameworks.

[3] https://www.eventstorming.com



Impact Mapping

On many software projects the developers only get the lower tiers of an impact map—what the business thinks they need and how they think the developers should achieve it. With an impact map, though, you can unwind their assumptions and nd out what they really want to achieve. And then you can use your technical expertise to suggest superior alternatives that they would never have thought of.

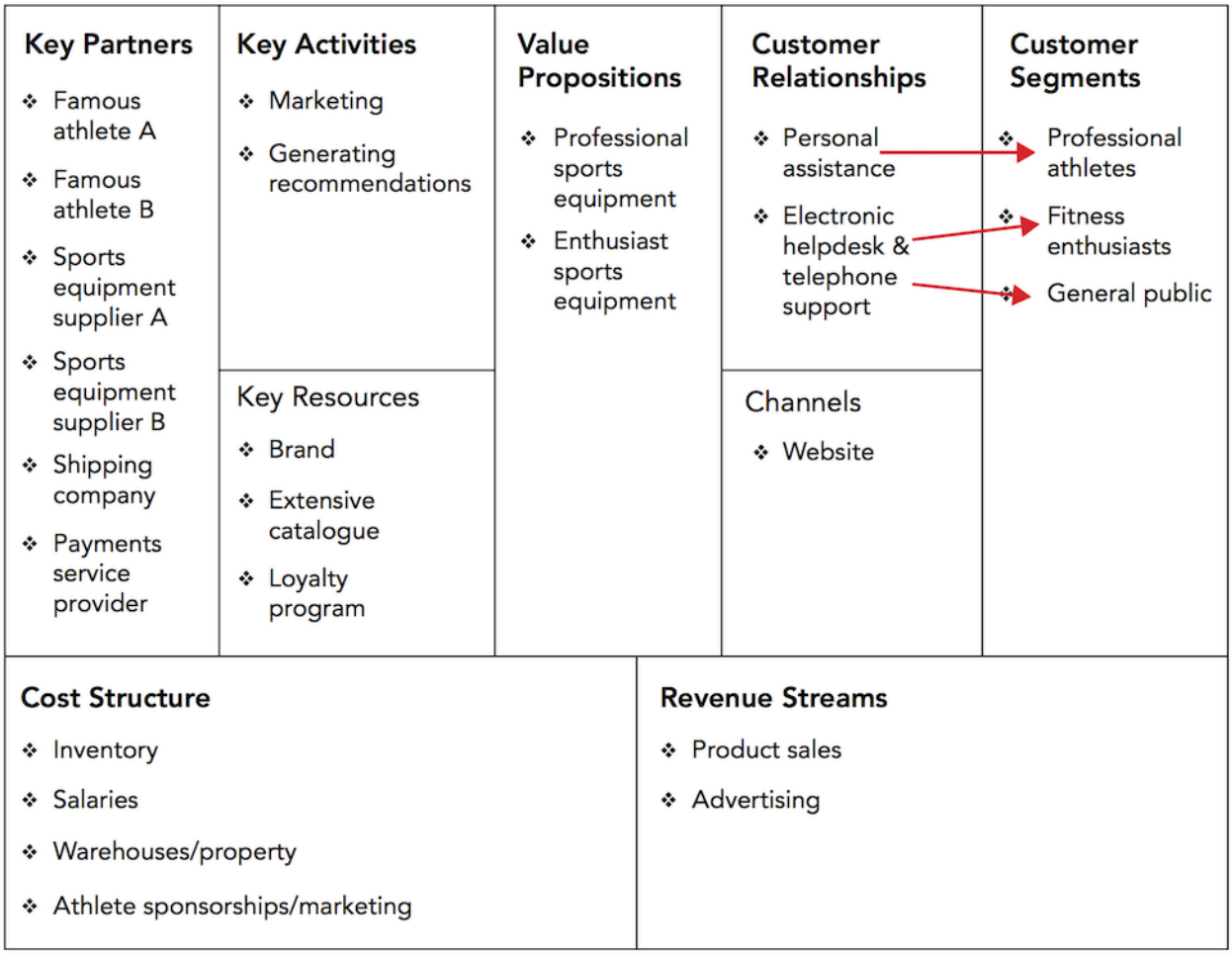
Some DDD practitioners rate impact mapping very highly, both when applied with DDD or in isolation. You are highly encouraged to investigate impact mapping by browsing the website[4] or picking up a copy of the book: "Impact Mapping," by Gojko Adzic[5].

[4] http://www.impactmapping.org/
[5] https://amzn.to/2AXdEgC

Understanding The Business Model

A business model contains lots of useful domain information and accentuates the fundamental goals of a business. Unfortunately, very few developers take the time to understand the business model of their employers or even to understand what business models really are.

One of the best ways to learn about a company's business model is to visualize it using a Business Model Canvas; a visualization technique introduced by Alexander Osterwalder and Yves Pigneur in their influential book, "Business Model Generation" is highly recommended and very accessible reading for developers. A Business Model Canvas is extremely useful because it breaks down a business model into nine building blocks, as shown in image, which illustrates an example Business Model Canvas for an online sports equipment provider.



Business Model Canvas

Understanding the nine building blocks of a business model tells you what is important to the business. Key information like: how it makes money, what its most important assets are, and crucially its target customers. Each of the sections of a business model is introduced below. For more information, the "Business Model Generation" book is the ideal learning resource.

- Customer Segments—the different types of customers a business targets. Examples include niche markets, mass markets, and business-to-business (b2b).
- Value Propositions—the products or services a business offers to its customers. Examples include physical goods and cloud hosting.
- Channels—how the business delivers its products or services to customers. Examples include physical shipping and a website.
- Customer Relationships—the types of relationships the business has with each customer segment. Examples include direct personal assistance and automated electronic help facilities.
- Revenue Streams—the different ways the business makes money. Examples include advertising revenue and recurring subscription fees.
- Key Resources—a business's most important assets. Examples include intellectual property and important employees.
- Key Activities—the activities fundamental to making the business work. Examples include developing software and analyzing data.
- Key Partnerships—a list of the business's most significant partners. Examples include suppliers and consultants.
- Cost Structure—the costs that the business incurs. Examples include salaries, software subscriptions, and inventory.

Armed with the information presented by a Business Model Canvas you will be empowered to ask meaningful questions of domain experts and help to drive the evolution of the business—not just the technical implementation. The small effort of finding and understanding your employer's business model is well worth it.

Applying The Theory Of Constraints And Systems Thinking

The theory of constraints (TOC) is a management paradigm introduced by Eli Goldratt that states that a system will be limited to achieving its goals by a few small constraints. Therefore focus and effort should be aimed at removing these constraints to the system above anything else. This will ensure that any output of effort results in maximum outcome to the business goal - more often than not gaining money. In very simple terms, identify the bottleneck that restricts the production of business value and remove.

TOC has a five step process:

- Identify the constraint. Understand the key issue that is causing a bottleneck or preventing your business generating value
- Determine how to eliminate the constraint. Focus all efforts at removing the constraint as fast as possible
- Subordinate everything else to the constraint. Don't be distracted by any other problems that are not contributing the constraint. Any other effort elsewhere is wasted effort. The constraint is the priority to resolve.
- Remove the constraint. Apply the solution to the constraint in order to remove it.

- Determine if the constraint has been removed. Sometimes you can push a problem down or up stream. If this is the case simply repeat the steps for the new constraint.

TOC is all about macro level over micro level thinking. It's not about optimising the link, its about optimising the entire chain or the system. To put into other words it's about optimizing your business to achieve its goals rather than a sub-department of service.

Why is this relevant to DDD you may ask? DDD has the domain or business at the heart of its philosophy, in that design decisions should be about driving business value. If you don't understand the constraint that is stopping the business from producing value then you can't make a good decision on where to focus effort. The constraint could be your core domain or it could be a problem in a supporting domain that affects your core domain. Improving anything other than the constraint may have value but it won't be the most valuable thing to do. It's about removing silo mentally and making sure that teams understand the bigger picture so that they can apply effort in the best possible place.

As an example, imagine you are part of a team that has been tasked with increasing online transactions by 10%. In order to gain alignment on where to focus you could apply TOC to determine what is the constraint for people checking out. Is it the registration page? If you see a high amount of dropouts you could look for a solution and correct the problem. Then you would look to see what the next big constraint to checking out is. It could be that the checkout flow is fine but by actually increasing the range of products available you would achieve 10% increase in transactions, all being equal.Alternatively if you optimised payment it would have little impact, even if done with great effort and skill, as the constraint is further up the flow.

Principle 2: Collaborate Towards A Solution

By gaining alignment on the real business constraint or opportunity you will be able to progress from having a shallow understanding of your problem domain but a good grasp on requirements to a deep understanding of business needs with an eye on opportunities that you can enable. Remember, people will tell you what they want rather than what is needed, the difference is subtle but leads to very different outcomes. Therefore don't sit and wait for requirements - offer solutions.

Start with a big picture understanding of the vision of the solution so that everyone has an alignment point. Highlight the important areas and those that need only to be good enough. Facilitate collaborative workshops on model design and problem simplification. You need to actively participate in solution design away from the computer. Contribute to your own requirements, don't leave it to others.

I often wonder if the titles of roles are somewhat self fulfilling. When I first started out in the world of IT my first job title was Analyst Programmer. The definition of an analyst is an individual who performs analysis of a topic and Programmer is a person who writes computer programs. Therefore half of my job was to understand about problems and the other half was to translate that understanding into language a computer could understand. Titles such as software developer

and engineer focus solely on the technical side and assume that the solution is refined enough that it can be turned into a set of requirements. The reality as we all know is seldom so.

A focus purely on the technical side highlights a bigger problem: when designing software for systems with complex logic, typing code will never become a bottleneck. The code is an artifact of developers and domain experts working together and modeling the a solution for the problem domain. The code represents the end of the process of that collaboration and discovery. A developer's job is to problem solve, and problem solving is sometimes easier done away from the keyboard in collaboration with domain experts. In the end, working code is ultimately the result of learning and an understanding of the domain.

Show Passion For The Problem Domain

> "Ignorance is the single greatest impediment to throughput." Dan North

Developers are fantastic at educating themselves on technology and project methodologies; however, decomposing a problem and being able to distill what is important from what is not will enable a good developer to become a great one. Ensure you spend as much time working to understand the problem space as you do in the solution space.

Just as a useful model is derived over a series of iterations, so too must a problem space be refined to reveal the true intent behind the original vision. Listening and drawing the why as well as the what and when from stakeholders is a skill that developers should practice just as they practice coding katas.

Software development is a learning process, so is the quest to reveal deep insights in your problem domain. Software development is about understanding; software that works to solve a business problem proves that understanding. Curiosity and a genuine desire to learn about a problem space is vital to produce good solutions and meaningful software. If you want to be good at anything, you need to practice, practice, practice. If you want to be a great developer rather than a good one, you need to show passion for the problem and commit to understand your domain.

To apply the principles of DDD, you need a driven and committed team—a team committed to learning about its craft and the problem domains it works in. Passion lies within all of us, and if you feel value in the practices of DDD, it is up to you to inspire your team and become an evangelist. Passion is contagious; if you commit to spend time with your domain experts to understand at a deeper level and can show how this results in a more expressive codebase then your team will follow.

Have passion for the problem space, Be proactive, be curious - this will make it easier to find solutions and more importantly opportunities. You cannot begin to be useful to solve a problem or look for opportunity if you don't really understand the domain you are working in to a sufficiently deep level. You are a professional. You are paid well. You have a responsibility to understand the domain you are in.

Learn To Facilitate Solution Exploration

Making sense of a complex problem domain in order to create a simple and useful model requires in-depth knowledge and deep insight that can only be gained through constant collaboration with the people that understand the domain inside and out. Complex problem domains will contain a wealth of information, some of which will not be applicable to solving the problem at hand and will only act to distract from the real focus of your modelling efforts. Knowledge crunching is the art of distilling relevant information from the problem domain in order to build a useful model.

Domain knowledge is key, even more so than technical know-how. Teams working in a business with complex processes and logic need to immerse themselves in the problem domain and, like a sponge, absorb all the relevant domain knowledge. This insight will enable teams to focus on the salient points and create a model at the heart of their application's code base that can fulfill the business use cases and keep doing so over the lifetime of the application.

Facilitation skills are vital in order to efficiently and effectively distill knowledge from domain experts. You don't need to be the smartest person but you do need to be able to collaborate and facilitate with smart people to gain deep insights. Only by asking the right questions can decisions be made, knowledge can be shared and a solution can be revealed and agreed upon. As covered in the Principle 1: Gain Agreement on the Problem Alberto Brandolini has done a tremendous amount of work in this area under the practice of event storming and big picture visual modeling. But this can also be used to focus on solution exploration at a lower level.

The key to facilitation is about guiding people not to an answer but to a shared understanding and empowering them to take responsibility. It's not to lead a group and make good decisions, it is to make sure good decisions get made. Workshops and collaborative working must enable a platform that accepts different points of views. No one should have the authority on a good idea, and no suggestion is stupid

Deliberate Discovery At The Constraint

Dan North, the creator of BDD, has published a method for improving domain knowledge called deliberate discovery. Instead of focusing on the framework of agile methodologies during planning and requirement gathering stages, such as the activities of planning poker and story creation, you should devote time to learning about areas of the problem domain that you are ignorant about. A greater amount of domain knowledge will improve your modeling efforts. At the start of a project teams should make a concerted effort to identify areas of the problem domain that they are most ignorant of to ensure that these are tackled during knowledge-crunching sessions. Teams should use knowledge-crunching sessions to identify the unknown unknowns, The parts of the domain that they have not yet discovered. This should be led by the domain experts and stakeholder who can help the teams focus on areas of importance, such as an identified constraint or bottleneck, and not simply crunching the entire problem domain. This will enable teams to identify the gaps in domain knowledge and deal with them in a rapid manner.

Don’t Ask For Requirements, Look For Impact Opportunities

> “Opinion is really the lowest form of human knowledge. It requires no accountability, no understanding. The highest form of knowledge… is empathy, for it requires us to suspend our egos and live in another’s world. It requires profound purpose larger than the self kind of understanding.” — Bill Bullard

Be wary of business users asking for enhancements to existing software, because they will often give you requirements that are based on the constraints of the current systems rather than what they really desire. Ask yourself how often you have engaged with a user to really understand the motivation behind a requirement. Have you understood the why behind the what? Once you share and understand the real needs of a customer, you can often present a better solution. Customers are usually surprised when you engage them like this, quickly followed by the classic line: “Oh, really? I didn’t know you could do that!” Remember: You are the enabler. Don’t blindly follow the user’s requirements. Business users may not be able to write effective features or effectively express goals. You must share and understand the underlying vision and be aware of what the business is trying to achieve so you can offer real business value.

Requirements assume someone has all the answers - this is a massive fallacy. We have progressed passed the stage of simply getting requirements from people. We now should be able to partner with business counterparts in order to remove problems or introduce them to the art of the possible and the endless opportunities technology can enable. Remember be accountable, don’t be lazy - don’t delegate solution design to the business. It’s not someone else’s problem, it’s yours. Domain experts are not system experts. Learn and discover with them. I have often found it the case that domain experts don’t necessarily know how current system works, or should work. They are domain experts, experts in their fields, but not business process engineers, not systems experts.

Focus On The Most Interesting Parts

> “There is no compression algorithm for experience” -Andy Jassy, CEO AWS

The collaboration between the business and the development team is an essential aspect of DDD and one that is crucial to the success of a product under development. However, it is important to seek out those who are subject matter experts in the domain you are working in and who can offer you deeper insight into the problem area. DDD refers to these subject matter experts as domain experts. The domain experts are the people who deeply understand the business domain from its policies and workows, to its nuisances and idiosyncrasies. They are the experts within the business of the domain; they will rarely, if ever, have the title of domain expert. Instead, look for the product owners, users, and anyone who has a great grasp and understanding for the domain you are working in regardless of title.

When picking scenarios to model, don’t go for the low-hanging fruit; ignore the simple management of data. Instead, go for the hard parts—the interesting areas deep within the core domain. The

opposite of this is bikeshedding - which is the art of spending a disproportionate amount of time and energy over an insignificant or unimportant detail of a larger problem. The term comes from a story of a meeting to discuss the development of a nuclear power plant in which the majority of the time is spent on the design of the bike shed as that is the only part that everyone is able to understand.

Therefore focus on the parts of the product that make it unique; these will be hard or may need clarification. Time spent in this area will be well served, and this is exactly why collaboration with domain experts is so effective. Using domain experts' time to discuss simple create, read, update, and delete (CRUD) operations will soon become boring, and the domain expert will quickly lose interest and time for you. Modeling in the complicated areas that are at the heart of the product is exactly what the principles of DDD were made for.

The Core Focus of PotterMore.com

Pottermore.com was the only place on the web where you can buy digital copies of the Harry Potter books. Like any e-commerce site, it allows you to browse products, store products in a basket, and check out. The core domain of the Pottermore site is not what the customer sees, but rather what he does not. Pottermore books aren't DRM-locked[6]; they are watermarked. This invisible watermark allows the books that are purchased to be tracked in case they're hosted illegally on the web. The core domain of the Pottermore system is the subdomain that enables this watermarking technology to deter illegal distribution of a book without infringing on the customer. (The customer can copy the book to any other of his devices.) This is what was most important to the business, what set it apart from other e-book sellers, and what ensured the system was built rather than being sold on iTunes or other e-book sellers.

Ensure Everyone Understands The Vision Of The Solution

A domain vision statement can be created at the start of a project to explicitly capture what is central to the success of the software, what the business goal is, and where the value is. This message should be shared with the team and even stuck up on a wall in the office as a reminder to why the software is being written.

Amazon's Approach To Product Development

Amazon has a unique approach when it comes to forming a domain vision statement called working backwards[7]. For new enhancements, a product manager produces an internal press release announcing the finished product, listing the benefits the feature brings. If the intended customer doesn't feel the benefits are exciting or worthwhile, the product manager refactors the press release until the feature offers real value for the customer. At all times, Amazon is focused on the customer and is clear about the advantage a new feature can bring before it sets out with development.

[6] http://www.futurebook.net/content/pottermore-finally-delivers-harry-potter-e-books-arrive
[7] http://www.quora.com/What-is-Amazons-approach-to-product-development-and-product-management

Principle 3: Ensure The Solution Solves The Core Problem

> "Technology can bring benefits if, and only if, it diminishes a limitation.", Eli Goldratt

An initial model is a product of exploration and creativity. However you should not stop at the first useful model you produce and you should constantly validate your model against different ideas and new problems. Experimentation and exploration fuel learning and vital 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. You need to challenge your assumptions and realign with the big picture. Ask yourself are you sure that you are focused on the core problem and have not been distracted by an interesting yet less valuable side problem? Is the solution cost effective and simple enough? If the solution is complex have we missed a chance to simplify the problem?

Focus Effort On Outcome Over Output

In an ideal world, quality software would always be top of your agenda; however, it's important to be pragmatic. Sometimes a new system's core domain could be first to market, or sometimes a business may not be able to tell if a particular idea will be successful and become core to its success. In this instance, the business wants to learn quickly and fail fast without putting in a lot of up-front effort.

The first version of a product that is not well understood need not be well crafted. There may be uncertainty if the endeavour will be invested in over time, and therefore you need to understand why speed of delivery could top initial supple design. However, if the product is a success and there is value in a prolonged investment in the software, you need to refactor to support the evolution; otherwise, the technical debt racked up in the rush to deliver starts to become an issue. It's a balancing act to produce code that will help you in the future against getting product out in front of customers for fast feedback.

Strive For Simple Boring Code

> "Simplicity is prerequisite for reliability.", Edsger W. Dijkstra

Teams that are aligned with the philosophy of DDD focus more on the bigger picture and understand where to put the most effort. They will not apply the same architecture to all parts of a solution, and they will not strive for perfection in areas of little value. They will trade isolated and working software for unnecessary elegance and gold plating. Only the core domains need to be elegant due to complexity or importance. This is not to say that all other code should be poorly written, but sometimes good is good enough.

Applying techniques designed to manage complex problems to domains with little or no complexity will result in at best wasted effort and at worst needlessly complicated solutions that are difficult to maintain due to the multiple layers of abstractions. DDD is best used for strategically important applications; otherwise, the deep knowledge gained during DDD provides little value to the organization.

When creating a system, developers should strive for simplicity and clarity. Software that is full of abstractions achieves little more than satisfying developers' egos and obscuring the reality of a simple codebase. Developers who aren't engaged with delivering value and are instead only focused on technical endeavors will invent complexity because they're bored by the business problem. This kind of software design can lead to frustration for teams in the future that need to maintain the mess of technical layers.

Don't let design patterns and principles get in the way of getting things done and providing value to the business. Patterns and principles are guides for you to produce supple designs. Badges of honor will not be given out the more you use them in an application. DDD is about providing value, not producing elegant code.

Keep your model simple and focused, and strive for boring plain code. Often teams quickly fall into the trap of overcomplicating a problem. Keeping a solution simple does not mean opting for the quick and dirty; it's about avoiding mess and unnecessary complexity. Use simplicity during code review or pair programming. Developers should challenge each other to ensure they are proving a simple solution and that the solution is explicitly focused only on the problem at hand, not just a general solution to a generalized problem.

The Jurassic Park Principle

> "Your scientists were so preoccupied with whether or not they could that they didn't stop to think if they should." Dr. Ian Malcom, Jurrasic Park

All problems are not created equal; some are complex and are of little business value, so it makes no sense to waste effort in finding automated solutions for them. Complex edge cases do not always need automated solutions. Humans can manage by exception. If a problem is complex and forms an edge case, speak to your stakeholder and domain expert about the value of automating it. Your effort could be better served elsewhere, and a human might better handle this exception. You can produce elegant and beautiful software but if it provides no value or misses the point then it is utterly useless.

Simple problems require simple solutions. Trivial domains or subdomains that do not hold a strategic advantage for businesses will not benefit from all the principles of DDD. Developers who are keen to apply the principles of DDD to any project regardless of the complexity of the problem domain will likely be met with frustrated business colleagues who are not concerned with the less important areas of a business. Talk to people about the opportunity cost before writing code and don't worry about not solving all the problems.

If we remind ourselves of the theory of constraints we should avoid local optimisations at expense of improving the system. You should ask yourself will your output produce meaningful outcomes -

should you expend lots of effort and energy on an area of the solution that does not impact the core domain?

You need to master the art of saying no. This is of course very difficult in practice, but it is worth it. Well meaning business counterparts may want you to optimise for their department at the expense of the system. Empathising with them and helping them understand how this distracts you from the overall goal will give you more time to focus on the strategic high value areas.

Build Subdomains For Replacement Rather Than Reuse

When developing models in subdomains try and build them in isolation with replacement in mind. Keep them separated from other models, legacy code, and third party services by using clean boundaries. By coding for replacement rather than reuse you can create good enough supporting subdomains without wasting effort on perfecting them. In the future they can be replaced by off-the-shelf solutions or can be rewritten as business needs change. Strive for good boundaries rather than perfect models. Boundaries are often harder to change than a model.

Simplify The Solution By Simplifying The Problem

Writing software is costly, code is expensive to produce and maintain. If you can solve a solution without code it's great. If you can limit what you output this is a good thing. Teams should be rewarded for developing simple solutions that enable business capability at a low actual cost and low opportunity cost. One way to do this is to change the problem. If you have a poorly understood or inefficient business process then applying a technical solution to it will simply create an expensive automated, but more complex process.

Don't be lazy and code around the problem. Sometime it's easier to code and add to domain complexity rather than change a business process as it's difficult to get the decision makers in a room. But you must do everything you can to simplify problems so that you can produce simpler solution models. Highlight an overly verbose or poor business process to your business counterparts so that are able to make changes. It's easy and lazy to not change processes with the excuse that it's been like that and it's established, especially when it requires complex code to work around it. We understand the value of code refactoring why not refactor the business? Don't accept mediocrity in business processes, highlight it at the root and speak to your business counterparts before trying to automate.

Challenge Your Solution

> "All good solutions have one thing in common: they are obvious but only in hindsight.", Eli Goldratt

When you are starting out on a new solution you are probably least knowledgeable about the domain. Paradoxically this is also the time when you will be making important decisions. Complex domains

are full of uncertainty, you need to be comfortable with this and with not knowing all the answers upfront. Remember a wrong choice is rather like an AB test, you chalk it up to experience and you have more knowledge on what doesn't work.

They say all models are wrong, it's just that some are more useful. Your initial model will be wrong, but don't get too hung up. Remember you need to love the problem not your initial solution. The process of learning more about the problem domain is achieved over many iterations and evolutions.. Your knowledge will grow, and to your solution, into something useful and appropriate.

Explore and experiment to reveal insights and offer new solutions. The result of tackling a problem from various angles is not the creation of a perfect model but instead the learning and discovery of concepts in the problem domain. This is far more valuable and leads to a team able to produce a useful model at each iteration.

Model Exploration Whirlpool

Eric Evans has created a draft document named the Model Exploration Whirlpool[8]. This document presents a method of modeling and knowledge crunching that can complement other agile methodologies and be called upon at any time of need throughout the lifetime of application development. It is used not as a modeling methodology but rather for when problems are encountered during the creation of a model. Telltale signs such as breakdowns in communication with the business and overly complex solution designs or when there is a complete lack of domain knowledge are catalysts to jump into the process dened in the Model Exploration Whirlpool and crunch domain knowledge.

The whirlpool contains the following activities:

- Scenario Exploring A domain expert describes a scenario that the team is worried about or having difficulty with in the problem domain. A scenario is a sequence of steps or processes that is important to the domain expert, is core to the application, and that is within the scope of the project. Once the domain expert has explained the scenario using concrete examples the team understands, the group then maps the scenario, like event storming in a visual manner in an open space.
- Modeling At the same time of running through a scenario, the team starts to examine the current model and assesses its usefulness for solving the scenario expressed by the domain expert.
- Challenging the Model Once the team has amended the model or created a new model they then challenge it with further scenarios from the domain expert to prove its usefulness.
- Harvesting and Documenting Significant scenarios that help demonstrate the model should be captured in documentation. Key scenarios will form the reference scenarios, which will demonstrate how the model solves key problems within the problem domain. Business scenarios will change less often than the model so it is useful to have a collection of important ones as a reference for whenever you are changing the model. However, don't try and capture every design decision and every model; some ideas should be left at the drawing board.

[8] http://domainlanguage.com/ddd/whirlpool/

- Code Probing When insight into the problem domain is unlocked and a design breakthrough occurs the technical team should prove it in code to ensure that it can be implemented.

Exploring Multiple Models

> "It's better to be approximately right than precisely wrong." —Eli Goldratt

Most teams usually stop exploring and jump to their keyboards when they arrive at the first useful model. Your first model will unlikely be your best. Once you have a good model, you should park it and explore the problem from a different direction. It's not about being right or wrong it's about moving forward, progressing and driving toward value. You will make wrong decisions but only from the luxury of hindsight so please don't beat yourself up - remember this is a learning process.

There is no such thing as a stupid question or a stupid idea. Wrong models help to validate useful ones, and the process of creating them aids learning. When working in a complex or core area of a product, teams must be prepared to look at things in a different way, take risks, and not be afraid of turning problems on their head. For a complex core domain, a team should produce at least three models to give itself a chance at producing something useful. Teams that are not failing often enough and that are not producing many ideas are probably not trying hard enough. When deep in conversation with a domain expert, a team should not stop brainstorming at the first sign of something useful. Once the team gets to a good place, it should wipe the whiteboard and start again from a different angle and try the what-if route of investigation. When a team is in the zone with an expert, it should stay there until it exhausts all its ideas.

Only stop modelling when you have run out of ideas and not when you get the first good idea. Once you have a useful model start again. Challenge yourself to create a model in a different way, experiment with your thinking and design skills. Try to solve the problem with a completely different model. Constantly refactor to your understanding of the problem domain to produce a more expressive model. Models will change with more knowledge. Remember a model is only useful for a moment in time; don't get attached to elegant designs. Rip up parts of your model that are no longer useful, and be willing to change when new use cases and scenarios are thrown at your design.

Principle 4: Optimize the Overall System

> "A system is never the sum of its parts. It is the product of the interactions of its parts", Russel Ackoff

When working on the solution to a problem teams must still keep an eye on the bigger picture. It is easy to become lost and distracted when you are in the detail and lose sight of what the goal is. We need to ensure that all parts of the decomposed solution are working for the greater good in a effective collaborative manner.

In large and complex domains, multiple models in context collaborate to fulfill the requirements and behaviors of a system or process. A single team may not own all of the various sub components of

a system, some will be existing legacy code that is the responsibility of a different team, and other components will be provided by third parties that will have no knowledge of the clients that will consume its functionality. Teams that don't have a good understanding of the different contexts within a system, and their relationships to one another, run the risk of compromising the models at play when integrating bounded contexts. Lines between models can become blurred resulting in a Big Ball of Mud if teams don't explicitly map and understand relationships between contexts.

The technical details of contexts within systems are not the only force that can hamper the success of a solution. Organizational relationships between the teams that are responsible for contexts can also have a big impact on the outcome. Often, teams that manage other contexts are not motivated by the same forces, or they have different priorities. For solutions to succeed, teams usually need to manage changes in these situations at a political rather than technical level, or as Nick Tune refers to it - the sociotechnical design.

What is important to understand is that it is not the individual components of a system that need to work, it is the system itself. Teams needs to learn to collaborate and agree to overcome any obstacles to implementation. To do this they must understand how they fit into to the system as a whole.

Strive for team autonomy

Multiple teams working together on a solution should be organised so that they are loosely coupled and as far as possible autonomous. Restricting the number of dependencies for a team will enable them to them to move faster. Ideally a team would also be aligned to a business capability (what the business does, rather than how it does it) that it is enabling and that contributes to the overall solution. In a perfect world the software team should be embedded in the business department that they are providing capability for rather than sit in a central IT org structure in order to develop a deeper understanding for their part of the domain.

Loosely coupled but highly cohesive teams can achieve autonomy if they understand the goal and have collaborated together on a solution. This causes alignment which enables autonomy. However be careful, without alignment loosely coupled teams can become silos and follow their own agenda and needs for their business department counterparts.

Boundaries are very important. Don't rush for structure or put concrete boundaries in before you really understand your solution space. Try to avoid precision in the first instance as boundaries and organisational design are harder to move down the line. Play with the model and reveal the linguistic and business capability ownership boundaries before implementing software boundaries. Have patience, don't force it or look for perfection, and don't get hung up if you are proved wrong and need to rethink. It's all valuable knowledge and experience.

Collaborate To Solve The Big Picture

> "A company could put a top man at every position and be swallowed by a competitor with people only half as good, but who are working together." W. Edwards Deming

Although having completely independent teams is a productivity win, it's important to ensure that communication between teams still occurs for knowledge and skill-sharing benefits. Ultimately, bounded contexts combine at run time to carry out full business processes, so teams need a big-picture understanding of how their bounded context(s) fit into the wider system. Established patterns for this problem involve having regular sessions in which teams share with other development teams what they are working on, how they have implemented it, or any technologies that have helped them achieve their goals. Another excellent pattern is cross-team pair programming. This involves moving a developer to a different team for a few days to learn about that part of the domain. You can spawn many novel approaches based on these two concepts of having group sessions and moving people around.

Collaboration is easier if you have alignment in what you are doing, this is why it's important to constantly align around the big picture so that everyone remembers the goal that they are separately contributing to. Context maps are important but not exclusive, process maps, event maps and event storming are ideal to visualise a shared understanding of system flow and ensure all teams are aligned by understanding the goal.

There are a plethora of collaboration tools in any organisation. Slack, skype, email, phone are a few. However with a wash of collaboration tools I often find that teams have forgotten how to talk to each other. It is vital to have strong relationships with teams that you depend on. We must also ensure we take personal feelings out of how we work and attack the problem. Its ok to be angry, just be angry at the problem not the person. Complex systems will have many moving parts and many teams, after all no man is an island. People relationships are as important as code relationships, therefore refactor your personal relationships as they are key to delivering effective solutions.

Identify Constraints In Delivering The Solution And Work Through Them

The context map, ever evolving, ensures that teams are informed of the holistic view of the system, both technical and organizational, enabling them to have the best possible chance of overcoming issues early and to avoid accidentally weakening the usefulness of the models by violating their integrity.

In complex systems there will be many dependencies. You should understand and work with others that own these dependencies to unblock flow for the big picture. The more you manage and, ideally, remove dependencies the easier your life will become.

In many ways, the communication between bounded contexts, both technical and organizational, is as important as the bounded contexts themselves. Information that context maps provide can enable teams to make important strategic decisions which improve the success of a solution. A context map is a powerful artifact that can bring new team members up to speed quickly and provide an early warning for potential trouble hot spots. Context maps can also reveal issues with communication and work within the business.

Understanding Ownership And Responsibility

Accountability and responsibility are other non-technical areas that can affect the delivery of a solution. Defining team ownership and management for subsystems that you need to integrate with is essential for ensuring changes are made on time and in line with what you expect. Context mapping is about investigation and clarification; you may not be able to draw a clear context map straight away, but the process of clarifying responsibility, explicitly defining blurred lines, and understanding communication flow while mapping contexts is as important as the finished artifact.

When it comes to responsibility the entire team needs to understand what they are responsible for and how it fits in the bigger picture. Each member must have deep knowledge of what they are doing, why, and the logic behind the method being used to implement the solution. Of course, teams need to subscribe to the principle of what they build they support, but greater than this is the ability to clearly articulate how what they are building contributes to the system. Too often a team lead simply delegates components of a solution to junior members who have little or no idea how their part contributes to the whole.

Identify The Grey Areas Of Business Process

The business processes that happen between and take advantage of bounded contexts are often left in no-man's-land without clear responsibility and clarity regarding their boundaries and integration methods. A context map, focusing on the nontechnical aspects of the relationships, can reveal broken business process flow and communication between systems and business capabilities that have degraded over time. This revelation is often more useful to the businesses that are able to better understand and improve process that spans across departments and capabilities. The insight can be used to identify the often gray area between contexts that govern business process and address the accountability void early to ensure that ownership is not simply assumed. These important orchestrational processes can often be devoid of responsibility from development teams and business ownership, but paradoxically are immensely important to business workflows and processes.

Principle 5: Be A Positive Influence On The Team

> "Complaining does not work as a strategy. We all have finite time and energy. Any time we spend whining is unlikely to help us achieve our goals. And it won't make us happier."
> — Randy Pausch, The Last Lecture

Domain-Driven Design isn't a silver bullet. Just as switching from an upfront waterfall approach to a more agile/XP project methodology didn't solve all your software development problems, opting to follow DDD won't suddenly produce better software. The common denominator in any successful project is a team of clever people who are passionate about what they are doing and who care about it succeeding. As they say, culture eats strategy for breakfast.

However much time you will spend in front of the computer it will dwarf the time you spend working and talking to others. Show respect, patience and humility when working with your peers and business counterparts. Understand different personality types and how different types interact in collaborative sessions. Work out how people learn - are they big picture people or detail people. Perfect the art of active listening and show empathy, these are very important traits to get on in this world let alone in software development.

Teamwork and communication are as essential as technical ability in order to deliver change in large or complex domains. Enthusiasm and the ability to learn, technical and non-technical subject matter, are also key skills required by team members. These softer skills have as much, if not more to do with delivering success than technical knowledge alone. As I have mentioned before we have moved away from IT being an order taker that are simply given a dense set of requirements and told to get on with it. We need to empathise not only with business colleagues, but with business strategy and overarching needs. You don't need rock stars, you need capable team players.

In some respects coding is the easy part, after all, the computer will only do what you tell it to do (well most of the time). Collaborating with and understanding different people with different backgrounds and different ways of working is hard. You need to be able to play well with others. Problem solving is a team sport, getting or giving help is not a sign of weakness but is essential for improving oneself to become an effective team player.

Lastly don't assume you know something, always be prepared to challenge your thinking and your opinions - you may be surprised on things you thought you knew. Do not be closed off to a new way of thinking or looking at problems from other people's point of views. Remember there is no right or wrong and it often has an awful lot to do with context.

To The Next 15 Years And The Continued Reinvention Of DDD

DDD is now an uncontrollable force, it is far larger than the sum of its parts. It is a huge ever growing community of progressive and professional problem solvers learning how they can offer more to their business. I am afraid that even if we were to cut the head of off Eric Evans the community and practitioners would still grow and continue to evolve Domain-Driven Design. Plus he is a nice chap and it would make an awful lot of mess.

In my opinion the future of DDD lies in the blending between non-technical and technical "business" people. Knowledgeable technical people will turn into business decision makers due to more focus on key underlying business problems. Those that embrace the business as much as they embrace technology will flourish as leaders and actively contribute to the strategies of their domains rather than simply being order takers.

Thank you Eric for a fantastic book and thanks for the thousands that he has inspired (and that have inspired him). Thanks to those that have inspired me and given so much for the community to help us become a little better each day. Lastly remember it's all about context so don't take what I

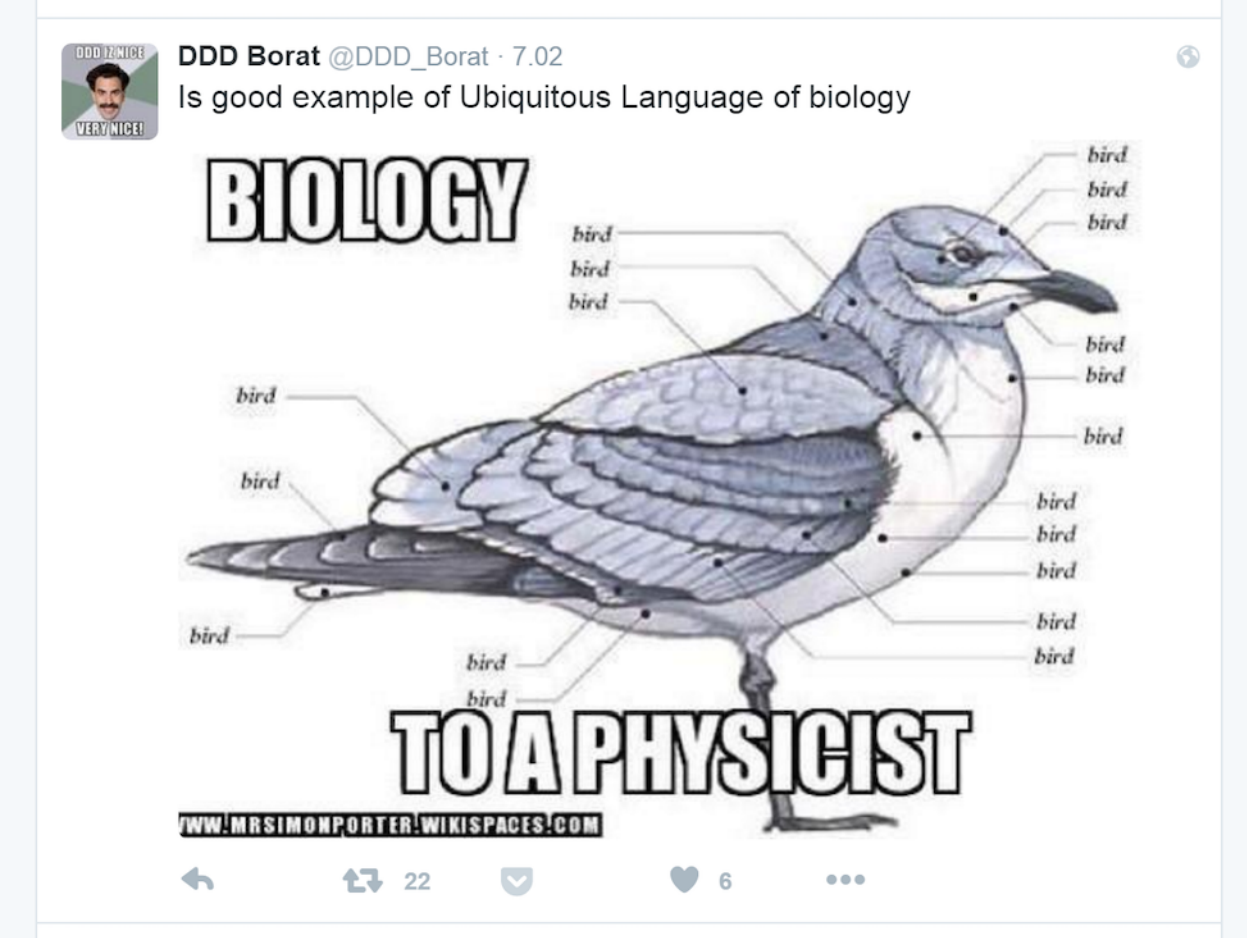
say, or anyone says, verbatim, understand things and learn how to leverage the experience in your own context. Always be the humble student, don't stop learning and never, never, never stop asking questions.

To DDD or not to DDD… What to do if your domain is boring? — Weronika Łabaj

I've heard countless times that DDD should ONLY be applied to complex, interesting domains. Even experts I deeply admire say that, so it must be true, right?

But I've never heard anybody explain what exactly makes a domain interesting or complex enough. Specifically, what makes it interesting or complex enough for the purpose of DDDifying it.

I think we've been asking the wrong question and this tweet by @DDD Borat summarizes my thinking perfectly:



A bird model for a physicist (by DDD Borat)

Before I explain what I mean exactly, let's have a quick look at two situations that shaped my opinion on the subject.

Example 1: Anemic domain model. Boring, Boring, Boring Line of Business App (BBBLOBA).

A few years ago I worked on an e-commerce system. We've basically implemented everything an online store does, starting from displaying catalogs, through managing promotions, calculating current prices, checkout process, through emails, payments integrations, to all the backend stuff that customers never see, like refunds or shipments integration.

The business logic was randomly divided between controllers, data access classes and stored procedures. There was no business logic layer whatsoever when I started there.

One day I was particularly frustrated. Looking at yet another property bag which was proudly (though inaccurately) called our "domain entity", I mumbled something about Anemic Domain

Model. My colleague overheard it. He sighed: “Yep, no wonder we have an anemic domain model here”, he said. “There’s no interesting business logic. That domain is just soooo boring.”

I wasn’t sure what to say. I’ve just spent a whole week investigating how refunds are calculated. I found a bug, which was rejected by our QA saying “It’s too complicated, I’m not raising it”. If that wasn’t an interesting piece of business logic, then I have no idea what else it was. So I only nodded my head in faked agreement and went back to work.

Example 2: Domain crunching applied. Boring, Boring, Boring Pet Project.

A while ago together with my boyfriend we got really pissed off with the mess which is called our personal finance. We simply had no idea where exactly our money went, how much we made and how much we spent.

We used Excel before, but that didn’t seem good enough anymore. Our finances got more complicated over time. Also our expectations got higher, because after a few years of tracking expenses, we knew what we wanted out of the app. So we did research, I checked dozens of apps and none of them fit our criteria. We could hack them to fit our needs and do some calculations outside of the app (probably in Excel).

Instead we did what any programmer would do in such a situation - we decided to write our own.

Now this surely is the most boring domain one could imagine, right? Apart from maybe a TODO list, it’s the most popular pet project ever. Every programmer at some point rolles out their own version of the expense tracker.

Over we’ve learned a lot about that domain.

The first insight was that in fact we have two main parts in our app, which are to large extent independent. Tracking and analyzing money. That fact impacted our model in significant way.

For example accounts are very useful for tracking. I can compare the actual total of one out of our 11 accounts and the data in the app. If some entries are missing, it’s helpful to know what kind of transaction it was. It narrows down the search in the online banking systems and helps with remembering where I last spent cash. Accounts are perfect for this purpose.

On the other hand I don’t care about accounts at the analysis stage. I don’t care if the specific payment was made from my personal account, our joint account, or my boyfriends wallet. Consequently, why in the world would I filter my expenses by account in reports? It gives me no actionable insights whatsoever. Adding such a feature would be a waste of time.

You might think “Isn’t it obvious? What’s the big deal?”. This is exactly what shows that we’re on the right track with modeling our domain. Afterwards it seems like the most obvious thing under the sun. But I can assure you it wasn’t that obvious when we started. Plus, most apps DO reports by account. Don’t ask me why would anybody use this. I have no idea, because I wouldn’t.

Then we hit our first problem. I'm self-employed, so I pay my own taxes and health insurance. It's not really an expense, in the sense that I can't do much to make it lower. So from the perspective of optimizing expenses it was only a noise in reports. Yet, I wanted to track it in some way.

At first we added a flag which indicated whether the specific expense should be included in analysis. But after a while we've realized we have a missing concept here - an income cost. We have other expenses that shouldn't be included in reports, but tracking my income costs on its own is actually very useful information! Making this concept explicit made our code easier and less error-prone.

The next challenge came from my boyfriend. He does a lot of business trips. He covers expenses during his trips himself, and then his employer reimburses the expenses in the next month's salary. He goes mainly to countries more expensive than Poland, so those expenses are relatively high. They completely ruined our analysis, and made tracking all expenses challenging. We still wanted to track those in the app, so we don't forget to check if the money was returned, but those weren't just normal expenses like groceries. That lead us to discovering another new concept - reimbursable transactions.

Since I occasionally lend money to family or friends we thought about explicitly modelling loans too. After giving it some thought we decided that using the same reimbursable transactions for business trips and loans is good enough. If we ever need more detailed distinctions between those two types of transactions, we can adjust our model.

We've made many more discoveries like those described above. We used the app, noticed that something feels wrong, discussed it and refined our model. With each new discovery, our model got cleaner and made more sense. Looking back we were surprised we didn't come up with all of this from the very beginning. It looks so obvious looking back!

Back to the DDD Borat

If you think about this, an e-commerce platform is way more complex and interesting system than a personal finance app. Or is it?

I think we've been asking the wrong question here. It's not about the domain per se. It's not that one domain is more interesting or more complex than the other. Being interesting is not an intrinsic characteristic of a particular domain.

What makes a particular domain interesting are problems you are solving.

In many cases it doesn't make sense to go with your "domain crunching" very deep. If you can get away with CRUD, then by all means, do CRUD. If you're a physicist then a bird is a bird is a bird, that level of abstraction would do just fine. It wouldn't work that well for a biologist though.

The other thing that makes a particular domain interesting is you.

Can you notice when your model is not good enough anymore? Do you know when it's time for a re-modeling session? Do you ever talk to your domain experts using their language? Are you truly interested in their problems and how to best solve them?

If not then no domain would be "interesting enough" to justify using DDD.

No excuses

In our case applying DDD concepts to the pet project really paid off. The app is easy to use, we make fewer mistakes using it than in the beginning. When it was a simple CRUD app, I messed something up at least few times a week, because I forgot what flags to set in what situation. Fixing my mistakes could take us anything between 10 min. to 1 hr. Now each concept is modeled explicitly and I have no doubts how to enter all transactions, even when I'm exceptionally tired. And we got incredibly useful insights from all this data, which impacted our life in many ways.

So don't let the popular myth hold you back. You can apply the so-called "strategic design" principles even in your pet project starting today.

Discovering Bounded Contexts with EventStorming — Alberto Brandolini

There's plenty of outcomes from a Big Picture EventStorming, the most obvious being the collective learning that happens throughout the workshop. However, learning is not the only one!

In this essay, we'll see how to leverage EventStorming to discover candidate bounded contexts in a complex domain.

Why Bounded Contexts are so critical

Among the many ideas coming with Domain-Driven Design, Bounded Contexts have been initially hard to grasp, at least for me. It took me a while to realize how powerful and fundamental this concept is.

In 2004, I was probably too good at building monoliths, or maybe I just hadn't seen anything different yet. In a few years, after seeing a few consequences of mainstream architectural practices, I radically changed my mind.

Now I consider *"getting the boundaries right"* the single design decision with the most significant impact over the entire life of a software project. Sharing a concept that shouldn't be shared or that generates unnecessary overlappings between different domains will have consequences spanning throughout the whole sociotechnical stack.

Here is how it might happen.

- A common concept (like the `Order` in an e-commerce web shop) becomes vital for several business capabilities, raising the need for reliability and availability, up to the unexplored limits of the CAP theorem, where buying more expensive hardware can't help you anymore.
- Security and access control get more complicated: different roles are accessing the same information, but *not exactly the same*, hence the need for sophisticated filtering.
- Changing shared resources requires more coordination: *"we have to be sure"* we are not breaking anyone else's software and plans. The result is usually more and more meetings, more trade-offs, more elapsed time to complete, and less time for proper software development.
- Since everybody now depends on 'the Order', be it a database table, a microservice or an Order Management System, changing it becomes riskier. Risk aversion slowly starts polluting your culture; necessary changes will be postponed.
- Developers begin to call the backbone software *"legacy"* with a shade of resentment. It's not their baby anymore; it's just *the thing that wakes them up in the middle of the night.*

- Adding refactoring stories to your backlog becomes a farce: since there is no immediate business value to be delivered, refactoring is being postponed or interrupted. Your attempts to explain "technical debt" to your non-technical colleagues always leave you disappointed.
- The time it takes to implement changes to the core of your system is now unbearable. Business departments stop asking changes in those areas and are implementing workarounds by themselves.
- Now your workplace isn't just that fun anymore. Some good developers that made it great are now looking for more challenging adventures.
- The business isn't happy either: delayed software evolution caused some business opportunities to be missed, and new players are moving on the market at a speed that is inconceivable with your current software.

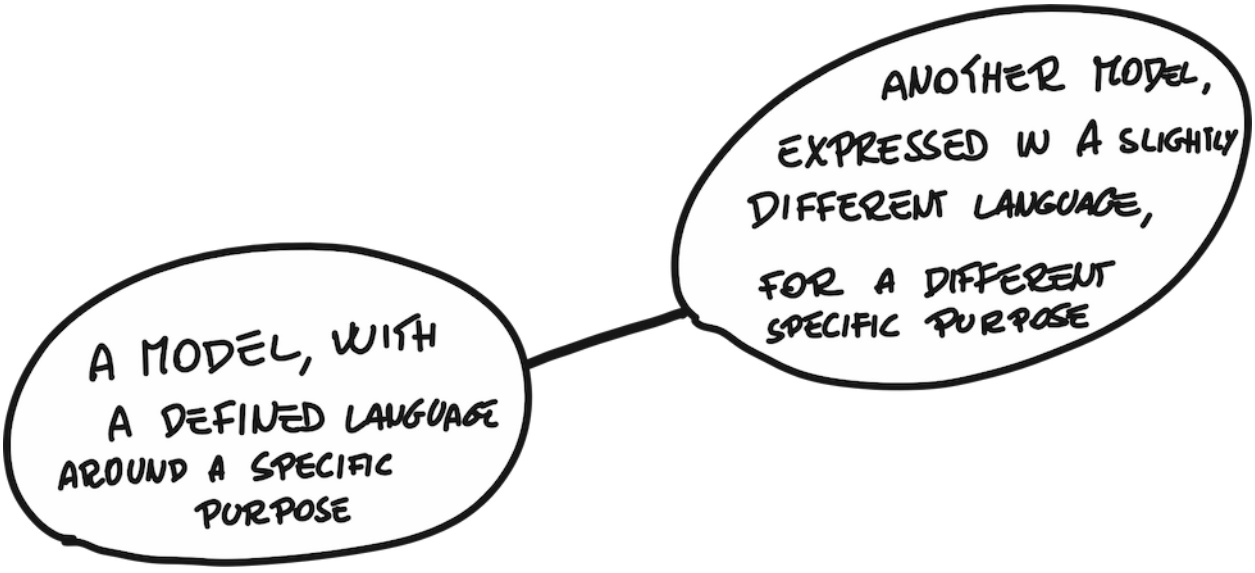
So here you are, yelling at the sky, asking yourself: *"What did we do wrong?"*

And a likely answer is: *"We didn't get the right context boundaries."*

Finding bounded contexts

Ideally, a bounded context should contain a model tailored around a specific *purpose*: the perfectly shaped tool for one specific job, no trade-offs.

Whenever we realize a different purpose is emerging, we should give a chance to a new model, fitting the new purpose, and then find the best way to allow the two models interact.



Two distinct purposes should map to two different models, inside different bounded contexts.

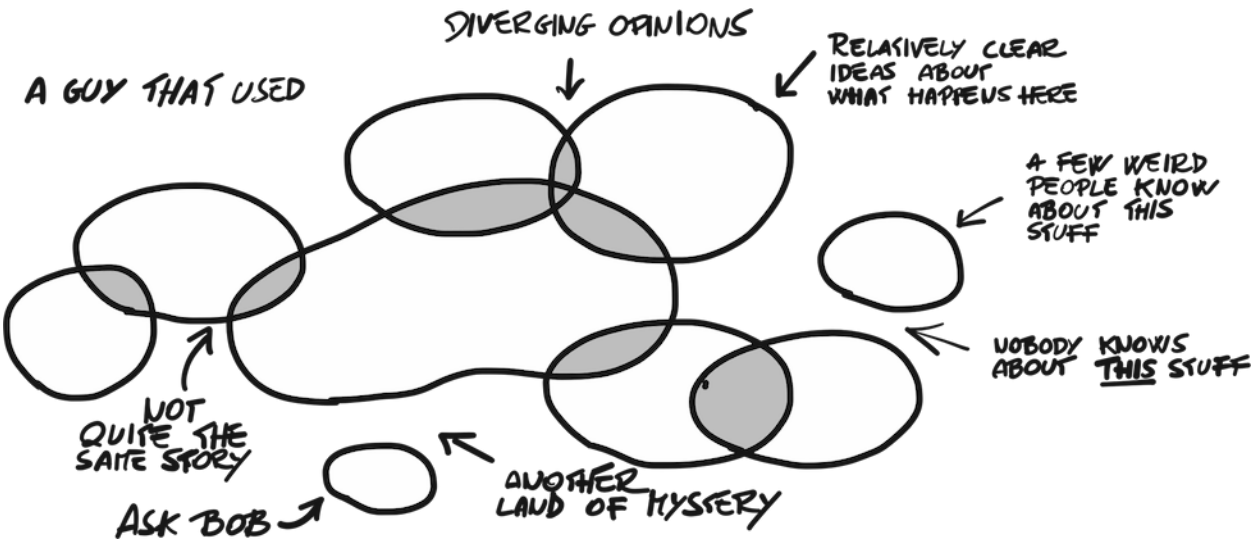
Unfortunately, *a single specific purpose* is not a very actionable criterion to discover boundaries in our model. The idea of 'purpose' is too vague to draw actionable boundaries: developers might be

looking for a clear, well-defined purpose, while business stakeholder might be a little more coarse-grained, like "I need an Employee Management Solution[9]."

In general, we can't assume the business side to know about bounded contexts. BCs are mostly a software development issue, and the learning loop about them will be closed in software development first.

Put in another way, the business stakeholders are not a reliable source of direct information about bounded contexts. *Asking* about bounded context will get you some information, but the answer can't be trusted blindly.

It's our job as software architects to discover boundaries in our domain, and this will be more an investigation on a crime scene than a *tick-the-checkboxes* conversation.



The knowledge distribution in an organization: a weird combination of knowledge and ignorance.

Nobody knows the whole truth. Let's stop pretending somebody can.

Enter EventStorming

EventStorming is a flexible workshop format that allows a massive collaborative exploration of complex domains. There are now several recipes[10], but the one that better fits our need to discover context boundaries is the **Big Picture EventStorming**: a large scale workshop (usually involving 15-20 people, sometimes more) where software and business practitioners are building together a behavioral model of a whole business line.

At the very root the recipe is really simple:

- make sure all the key people (business *and* technical stakeholders) are in the same room;

[9]Management is not a purpose. This is true on so many levels, but talking about bounded contexts this is especially true: *planning*, *designing*, *tracking*, *running*, *supporting*, *choosing* are more fine-grained purposes, that often require a model on their own.

[10]The best entry point to start exploring the EventStorming world is probably the official website.

- provide them with an unlimited modeling surface (usually a paper roll on a long straight wall plus some hundreds of colored sticky notes);
- have them model the entire business flow with **Domain Events** on a timeline.

In an EventStorming workshop, domain events — or just *events* — are not software constructs: they're short sentences written on a sticky note, using a verb at the past tense.

The simplest possible explanation of a domain event

With a little facilitation magic, in a few hours, we end up with a big behavioural model of the entire organization: something like the one in the picture below.

The output of a Big Picture EventStorming, on a conference organization scenario

A massive flood of colored sticky notes, apparently. But, as the adagio says, *it's the journey, not the destination*: the process of visualizing the whole business flow, with the active participation of all the key stakeholders, is our way to trigger critical insights and discoveries.

Structure of a Big Picture workshop

To make things clearer, let's see the main steps in the workshop structure, focusing mainly on the steps that provide critical insights to Bounded Contexts discovery.

Step 1. Chaotic Exploration

This is where workshop participants explore the domain, writing verbs at past tense on sticky notes (usually orange), and place them on the wall, according to an ideal timeline.

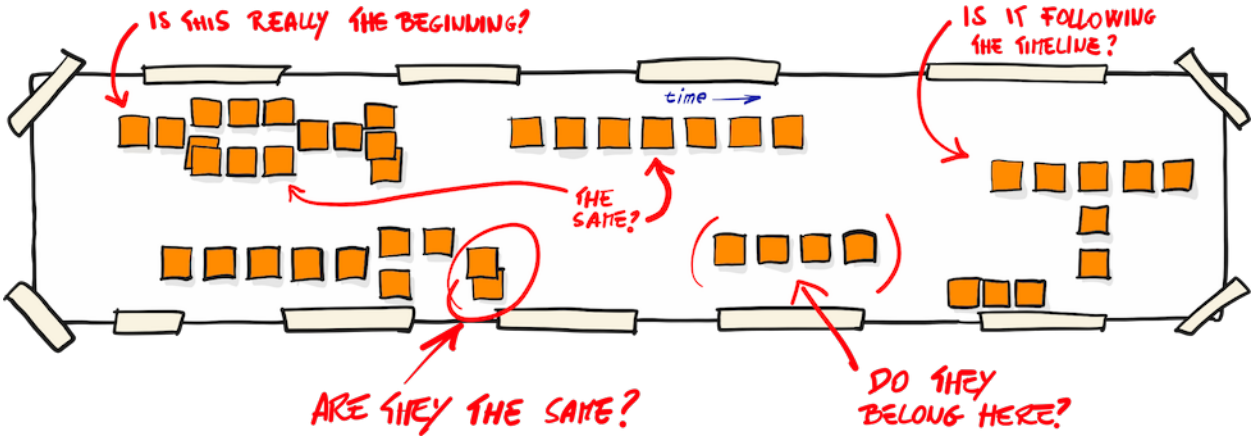
The key trick here is that nobody knows the whole story. Imagine we're exploring the domain

of conference organization[11]: there will be roles that know about strategy and market positioning, others specialized in dealing with primadonna speakers, plus a variety of other specialists or partners dealing with video coverage, catering, promotional materials and so on.

If you know one job well, you probably won't have time to know every other job at the same depth. This fragmentation of expertise is exactly what we're expecting to find in every business: local experts, masters of their silo, with variable degrees of understanding of the other portions of the business.

The more participants, the harder it is to follow a timeline: diverging perspectives and specialized view on what the business is really doing will usually lead to **clusters of locally ordered events, in a globally inconsistent whole**.

Far from perfect, but a good starting point. Now we see stuff.



The possible outcome of a chaotic exploration round in a Big Picture EventStorming.

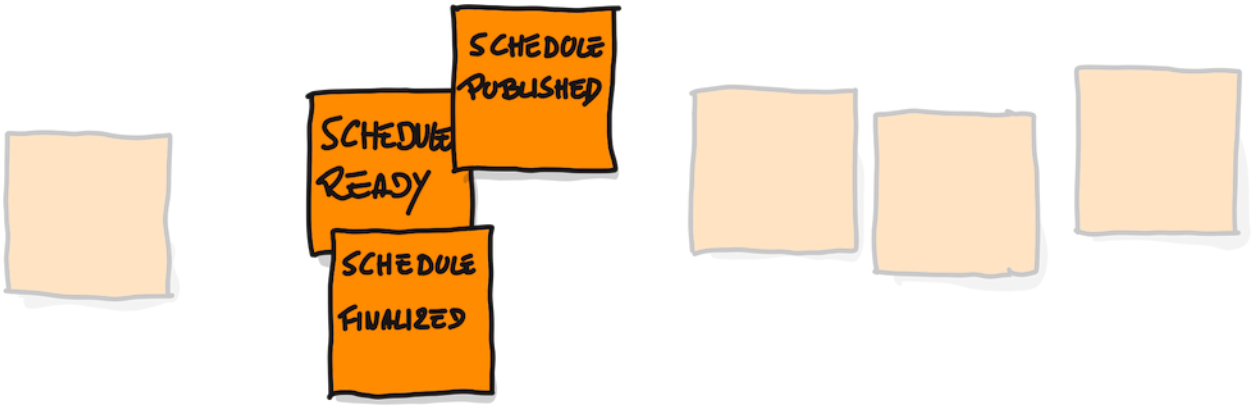
Moreover, this first step is usually *silent*: people will quietly place their braindump on the wall, wherever there's enough space available. Not so many conversations are happening. Next steps will be noisier.

Divergence as a clue

Actually, we want this phase to be quiet: people should not agree yet about what to write on sticky notes. The facilitator should make sure that there is plenty of markers and stickies so that everybody can write their interpretation of the flow independently, without influencing each other too much.

As a result, we'll end up with a lot of *duplicated* sticky notes, or *apparently duplicated* ones.

[11]Conferences are a little mess, but they are interesting because they often employ fewer people than the required roles: a small team is taking care of various activities spread around months, with a peak of intensity during the conference and the days before. The need for specialization is continuously at odds with the need of having to sync as few people as possible. At the same time, I've never seen two conferences alike, so I won't be revealing special trade secrets here.



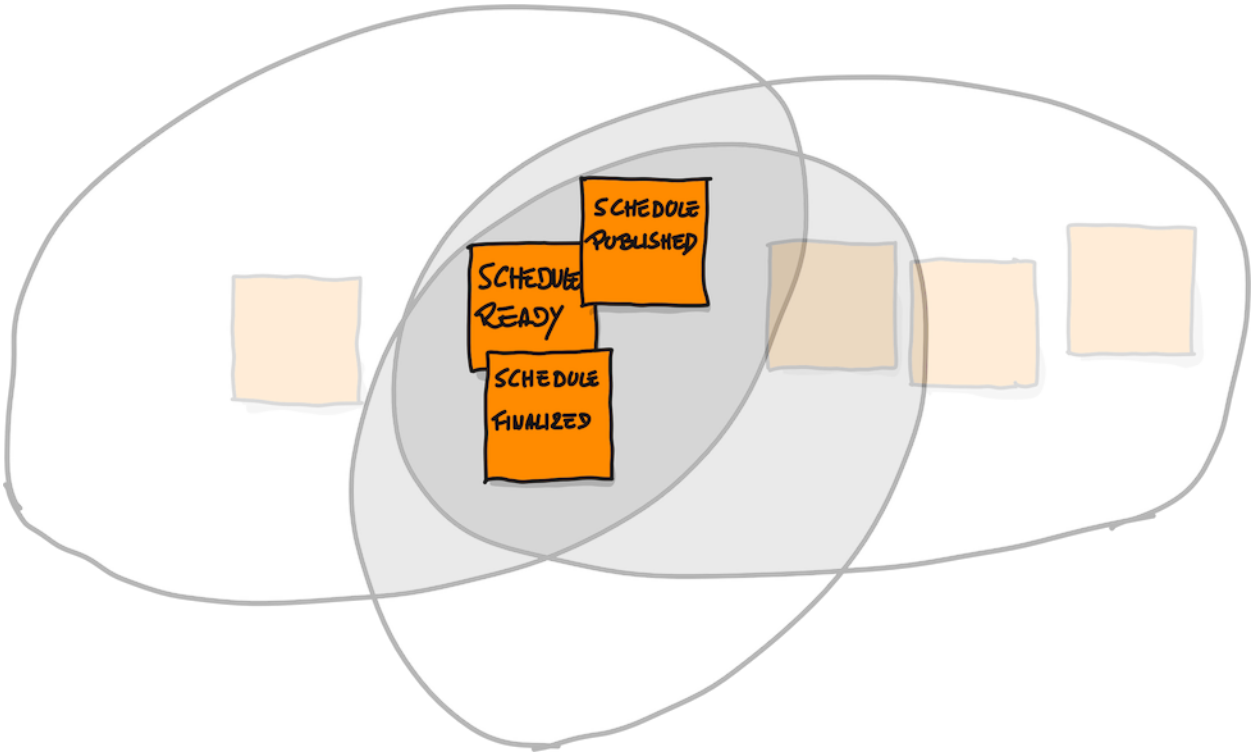
Different wordings are often a clue of different underlying meanings.

It's usually a good idea to resist the temptation to resolve those duplicates and find and *agree* on a single wording choice. Different wording may refer to different perspectives on the same event, hinting that this might be relevant in more than one Bounded Context, or that the two or more events aren't the same thing.

Getting back to our conference scenario, we might expect to have a few domain events referring more or less to the same thing, with different wording. Something like: `Schedule Ready`, `Schedule Completed`, `Schedule Published` and so on.

This discordance is already telling us something: this event (assuming or pretending there's only one event here) is probably relevant for different actors in the business flow.

That's cool! It might be a hint of multiple overlapping contexts.



Different meanings may point to different models, hence different contexts.

I didn't say *"bounded"* because the boundaries aren't clear yet.

Step 2. Enforce the Timeline

In this phase, we ask participants to make sure there is a consistent timeline describing the business flow from a beginning to an end.

It won't be that easy: there'll be parallel and alternative paths to explore. Even big-bang businesses, like conference organization, tend to settle on a repeating loop, usually repeating every year.

The need to come up with *one consistent view* of the entire business triggers **conversations** around the places where this view is not consistent. People start asking questions about what happens in obscure places, and they'll get an answer because we made sure the *experts are available!*

At the same time, we'll have diverging views about how a given step should be performed. Some conflicts will be settled — after all it's just a matter of having a conversation — other will be simply highlighted with a **Hot Spot** (usually a sticky note in the red spectrum, like magenta), to let the exploration flow.

[FIXME: picture of a hotspot.]

HotSpots clarify the underlying approach of our exploration: we're not here to *solve everything*, there is just no time for that. However, we can try to *visualize everything* including things that are unclear, uncertain or disputed.

The Big Picture EventStorming will deliver **the snapshot of our current collective level of understanding of the business** including holes and gaps.

Emerging structure

Simply talking about problems and moving sticks around won't do justice to the insights and discoveries happening during this phase. This is where participants often look for a more sophisticated structure, to sort out the mess they just created.

There are a few strategies to make the emerging structure visible. The most interesting for discovering bounded contexts are *Pivotal Events* and *Swimlanes*.

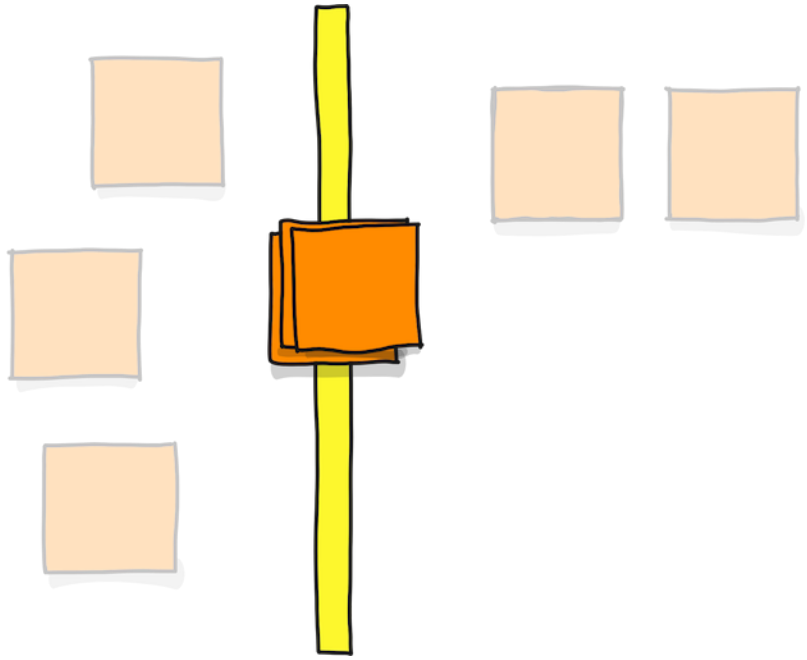
Using Pivotal Events

Pivotal Events are specific events which are particularly relevant for the business and mark a transition between different business phases.

In our conference organization scenario, we might spot a few candidates.

- `Conference Website Launched` or maybe `Conference Announced`: this is when you may start to sell tickets, but at the same time you can't easily withdraw anymore.
- `Conference Schedule Announced`: now, the speakers are officially in and ticket sales should start.
- `Ticket Sold`: on one side, here is a relevant business transaction, with money finally coming in; on the other one, we now have a *customer* and/or an *attendee* and a lot of specific communications that we'll have to manage.
- `Conference Started`: this is where attendees are expecting to get some value back from the ticket they purchased. Same goes for speakers looking for insights or contacts, and sponsors.
- `Conference Ended` looks like an obvious one. The party is over, let's deal with the aftermaths.

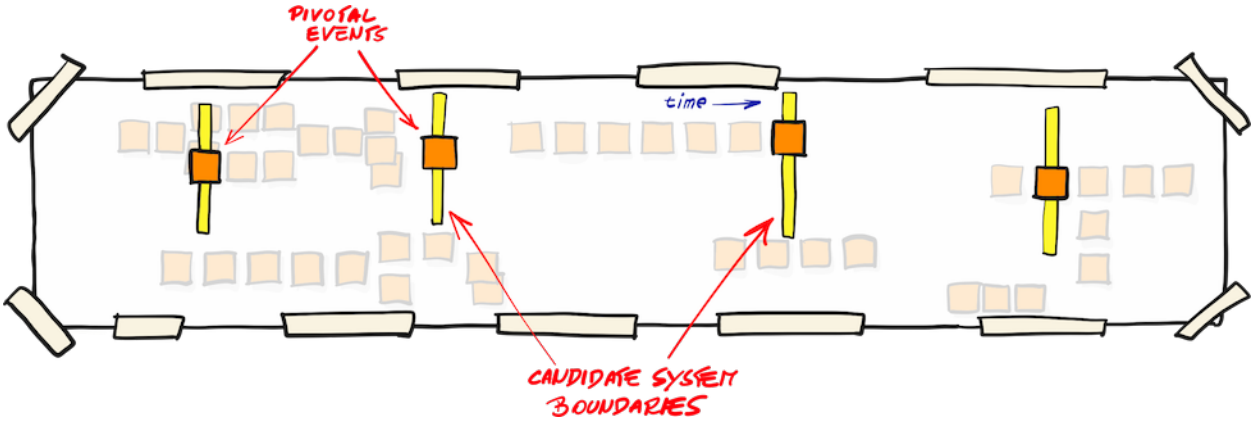
I usually mark the candidate events with colored tape, so that they're visible and we have a visible hint of distinct phases.



A piece of colored replaceable tape is my favorite tool for marking pivotal events.

It doesn't matter to pick the right ones, so I often keep this discussion short. I look for 4-5 candidate events that seem to fit that role. Mostly to sort out the internal events faster. We can still improve the model later if needed.

After highlighting pivotal events, sorting becomes a lot faster inside the boundaries, and a more sophisticated structure starts to emerge.



Pivotal Events are a great source of information.

Using Swimlanes

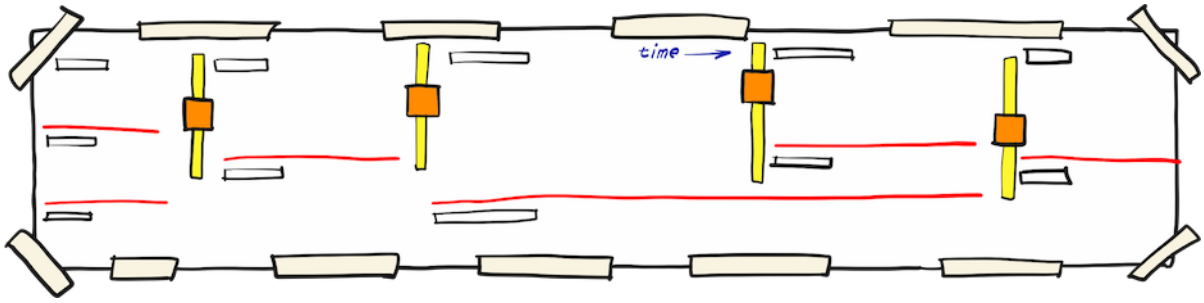
Even in the most straightforward businesses, the flow is not linear. There are branches, loops and things that happen in parallel. Sometimes the touch points between different portions of the flow are well-defined, like *billing* getting triggered only around the events of a sale, or maybe a cancellation.

Other times, the relationship can be more complicated: announcing some famous speaker can boost sales, sales can attract sponsors, sponsorships can allow organizers to afford the paycheck for more superstars speakers, which can trigger more sales and eventually trigger a venue upgrade.

Horizontal Swimlanes is a common way to structure portions of the whole flow. It usually happens after pivotal events, but there's no strict recipe here: the emerging shape of the business suggests the more effective partitioning strategy.

In our conference scenario, we can spot a few main themes that happen more or less in parallel: a *Speaker Management Flow* dealing with theme selection, call for papers and invitation, logistics and accommodation; a *Sales Management Flow* dealing with ticket sales, advertising, logistics and everything needed to welcome attendees, a *Sponsor Management Flow* managing the other revenue stream; and last but not least a lot of not so ancillary areas, from venue management, to catering, staffing, video recording and so on.

Our replaceable tape comes in handy also to give a name to these parallel flows. The underlying structure backbone will now probably look something like this:

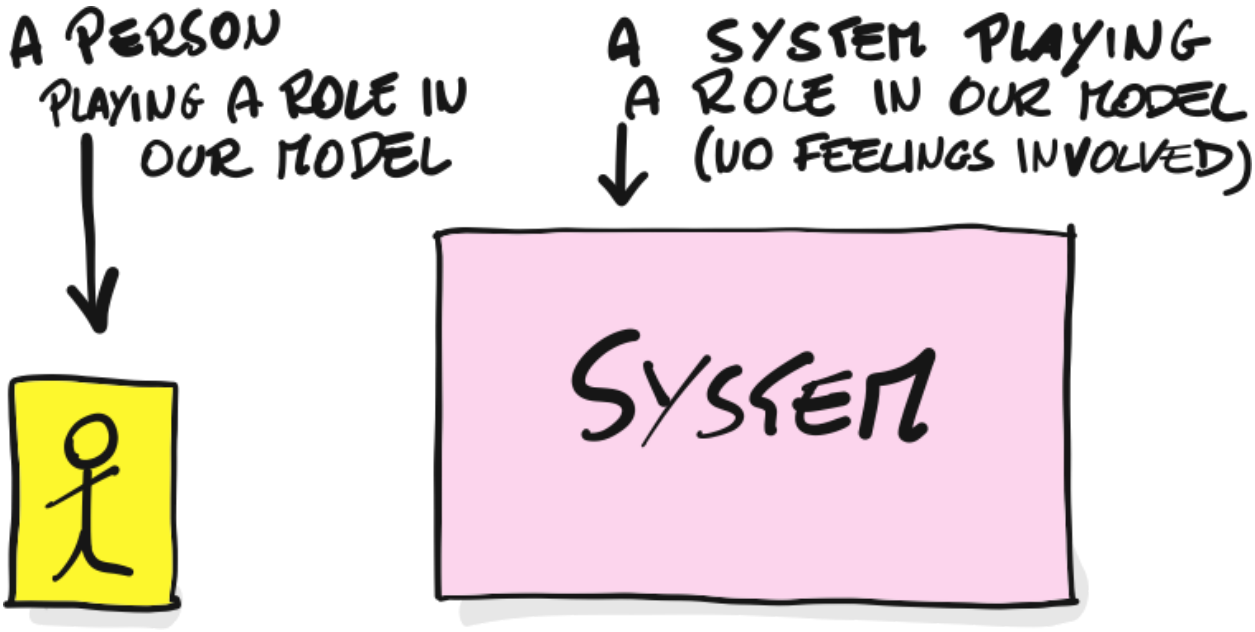


Pivotal Events and Swimlanes provide an emergent structure on top of the flow of Domain Events.

Everybody now sees the different *phases* of the business and the key relevant issues at every step.

Step 3. People and Systems

In this phase, we start exploring the surroundings of our business, explicitly looking for **people**: actors, users, personas, or specific roles in our system; and **systems**: from software components and tools to external organizations. Nothing excluded.



Here's the incredibly simple notation for people and systems.

Visualizing different actors in our system — in practice, we don't call them *actors* just *people* — helps to dig into the different perspectives. We might discover specific responsibilities and roles, or differing perceptions: all speakers are submitting talks, but the treatment can be different if we're talking about a superstar guest, invited as a keynote speaker, an expert or a newbie. A puzzled look during this exploration phase may end up in opening an entirely new branch or making different strategies more visible and readable.

Systems usually trigger a different type of reasoning. On the one hand, they make our boundaries explicit. We won't be able to deliver value in a vacuum, with self-contained software: tools, integrations, and collaborations are needed to complete the flow. Payment management systems, ticketing platforms, social media platforms, video streaming, billing software, and whatever comes to mind.

Systems can also be external entities like government agencies, or something as vague as "the weather" (which might be a severe constraint if you are organizing a conference in extreme regions, or if you're so lucky to get a snow storm the day before).

From the software perspective, an external system calls for some integration strategy (maybe our all-time favorite: the Anti-Corruption Layer), but from the business perspective, external systems often impose constraints, limitations or *alibis*, a free scapegoating service if anything goes wrong.

Step 4. Explicit Walkthrough

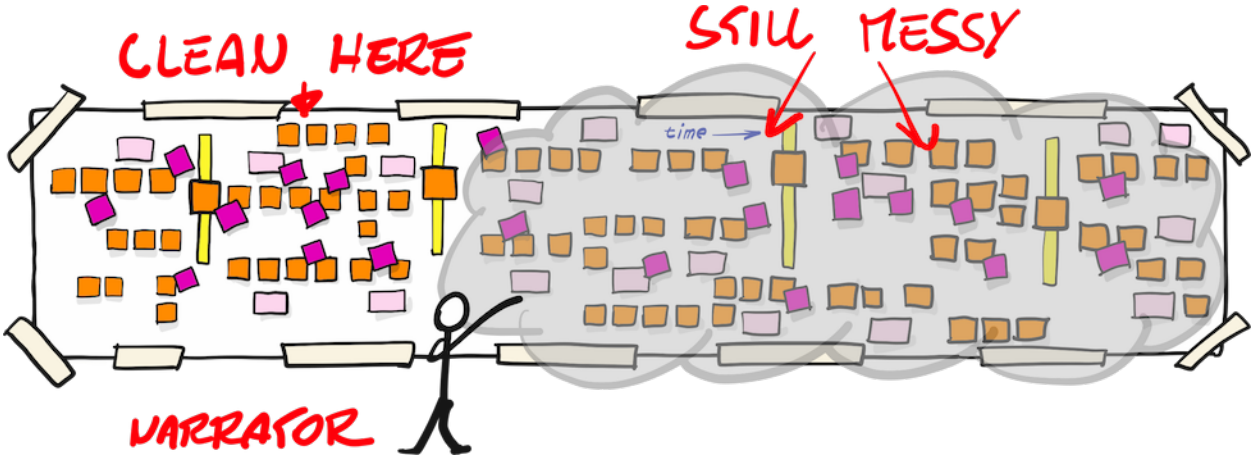
Every step triggers some clarification and prompts writing more events. Even if we added some structure, with pivotal events and swimlanes, and had some interesting conversation on the spot, the whole thing still feels messy. Probably because it is still messy.

It's now time to validate our discoveries, picking a *narrator* trying to tell the whole story, from left to right. Consistent storytelling is hard, in the beginning, because the narrators' brain will be torn apart by conflicting needs. Narrators will try to tell the story using the existing events as building blocks, but at the same time, they'll realize that what seemed *good enough* in the previous reviews is not good enough for a public *on stage* storytelling session.

Now our model needs to be improved, to support storytelling. More events will appear, others will be moved away, paths will be split and so on.

The audience should not be passive. Participants are often challenging the narrator and the proposed storytelling, eventually providing examples of corner cases and not-so-exceptional-exceptions.

The more we progress along the timeline, the more clarity is provided to the flow, while the narrator is progressing like a *defrag cursor*[12].



An explicit walkthrough round is our way to validate our understanding.

Extra steps

There are a few extra steps that might be performed now, usually depending on the context, that may provide more insights. I'll describe them briefly.

- We sometimes explore the **value** that is supposed to be *generated* or unfortunately *destroyed* in the business flow. We explore different currencies: money being the most obvious one, often to discover that others are more interesting (like *time*, *reputation*, *emotional safety*, *stress*, *happiness*, and so on).
- We explore **problems** and **opportunities** in the existing flow, allowing everyone to signal issues that didn't surface during the previous steps, or to make improvement ideas visible.
- We might challenge the status quo with an **alternative flow**: once the understanding of the current situation is settled, what happens if we change it? Which parts are going to stay the same and which ones are going to be radically changed or dismissed?

[12] If you're old enough to remember what *defrag* used to be. ;-)

- We might **vote the most important issue** to leverage the clarity of collective understanding into political momentum to do the right thing[13].

All this stuff, plus more, is usually awesome! But most of the information needed to sketch context boundaries is already available if you take a closer look.

Deriving this information from our workshop is now our job as software architects.

Homework time

Once the workshop is officially over, and participants left the workshop room, we can start talking software, *...finally!*

I used to draw context maps, as a way to *force myself to ask the right questions early in the project*, now I run EventStorming workshops, and I engage stakeholders in providing the right answers without asking the corresponding questions.

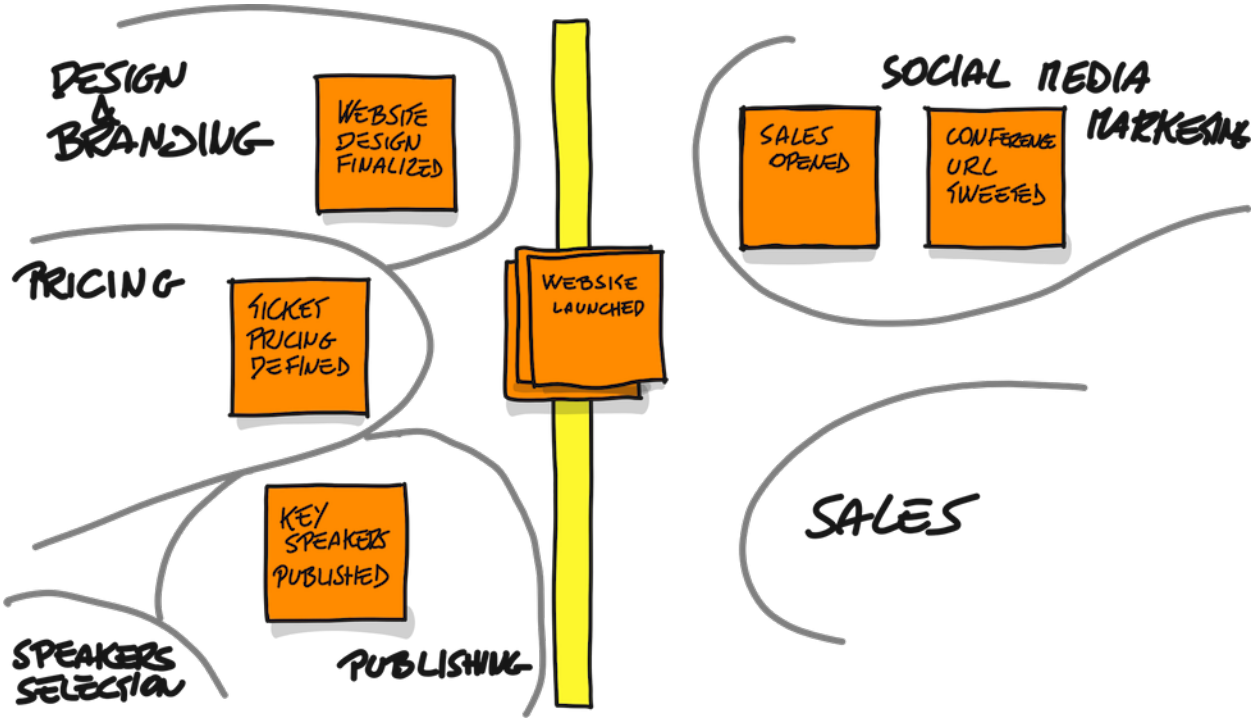
There's a lot of Bounded Context related info that comes as a byproduct of our discussion; we only need to decipher the clues. So, here we are with some heuristics[14], that may come in handy.

Heuristic: look at the business phases

...or like detectives would say: *"follow the money!"* Businesses grow around a well-defined business transaction where some value — usually money — is traded for something else. Pivotal events have a fundamental role in this flow: we won't be able to sell tickets online without a website, everything that happens before the website goes live is *inventory* or *expenses*, we can start making money only after the `Conference Website Launched` event.

Similarly, after `Ticket Sold` events, we'll be the temporary owners of attendees' money, but they'll start to get some value back only around the `Conference Started` event. But the tools and the mental models needed to *design* a conference, are not the same tools needed to *run* a conference.

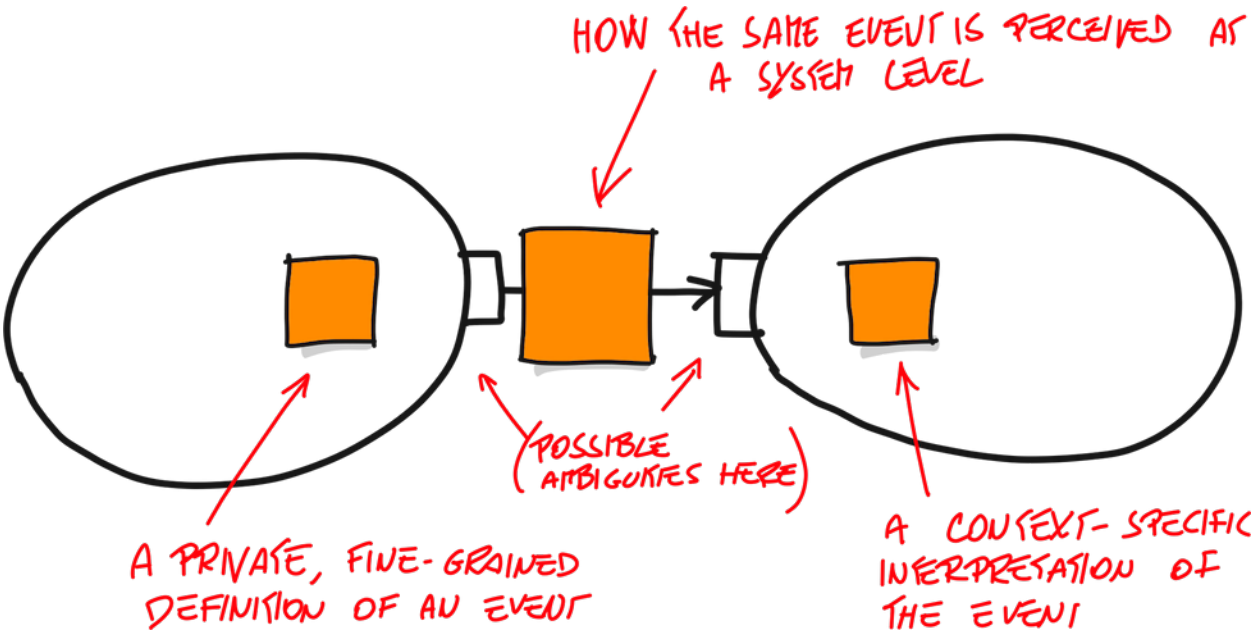
[13] This area might be what your *Core Domain* looks right now.
[14] I might have used the word *'heuristic'* here only to make Mathias Verraes happy.



A zoom-in into a pivotal event

Interestingly, boundary events are also the ones with different conflicting wordings. Here is where the perception of bounded contexts usually overlaps. A key recommendation here is that *you don't have to agree on the language!* There's much more to discover by making disagreements visible.

Moreover, keep in mind that when two models are interacting, there are usually *three* models involved: the internal models of the two bounded contexts and the *communication model* used to exchange information between them.

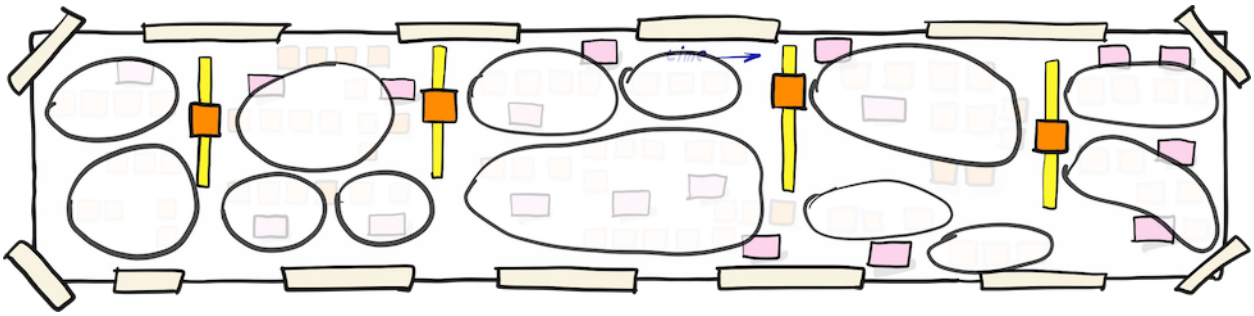


Two interacting models usually mean three languages.

A simple example: I am trying to communicate with my readers using the English language, but I am not a native English speaker. My internal reasoning model is sometimes English too, and sometimes Italian. But readers shouldn't be able to tell (I hope). At the same time, this text is not intended for British and American people only, every reader will translate into their mental model, possibly in their native language.

In general, **different phases** usually mean **different problems**, which usually leads to **different models**.

Pivotal Events are usually part of a more general *published language* shared between the different parties.



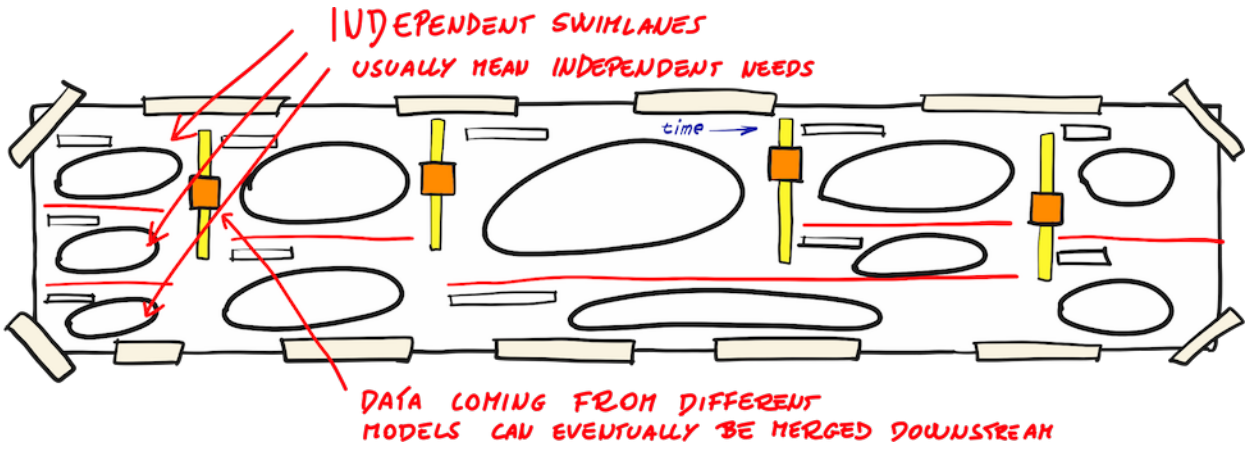
Emerging bounded contexts after a Big Picture EventStorming.

The picture above shows more or less what I am seeing when looking at the flow with Bounded Contexts in mind.

Heuristic: look at the swimlanes

Swimlanes often show different paths that involve different models.

Not every swimlane is a Bounded Context, sometimes they're just an *if* statement somewhere, but when swimlanes are emerging for the need to highlight an independent process, possibly *on a different timeline*, then you might want to give a shot to an independent model.

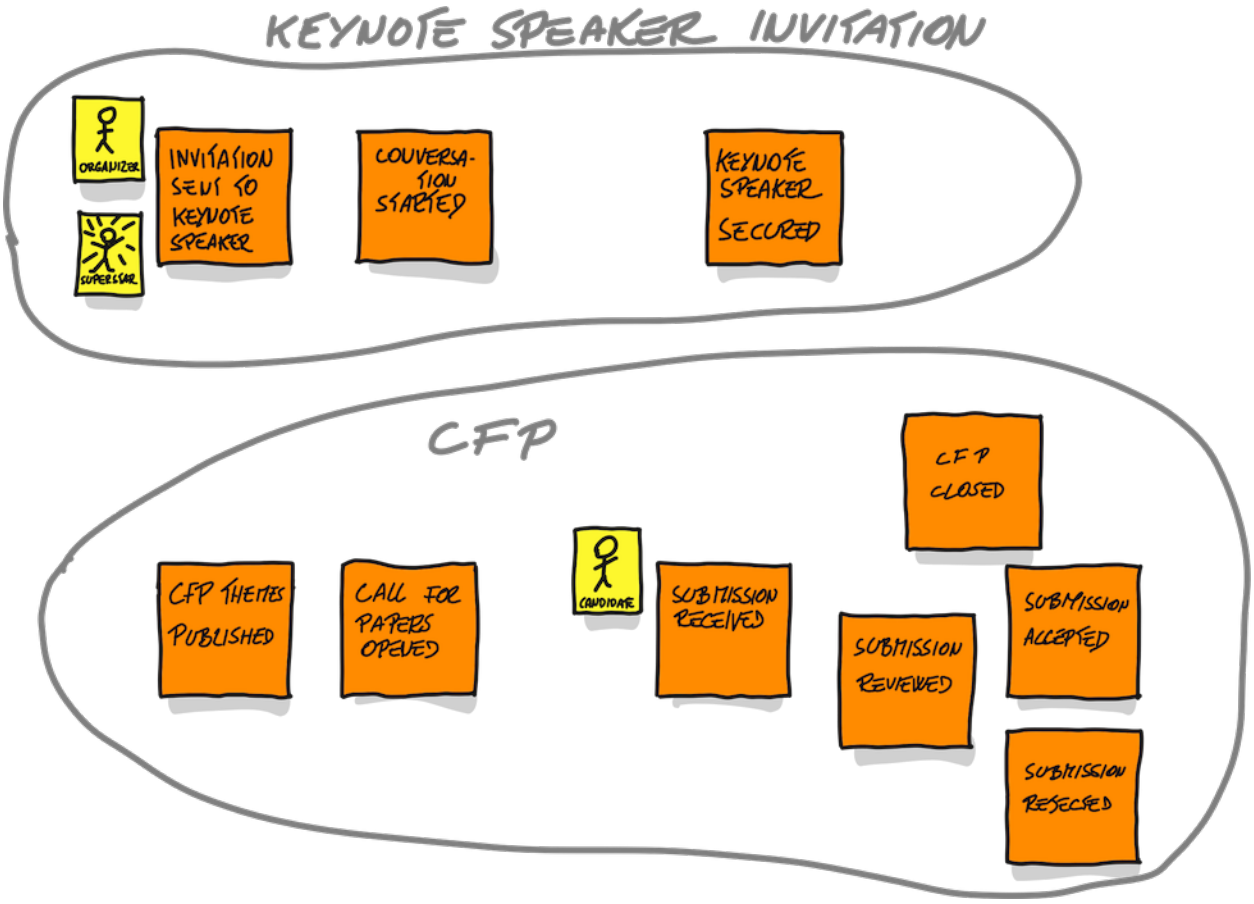


Swimlanes are usually a reliable clue for possible different bounded contexts.

Heuristic: look at the people on the paper roll

An interesting twist might happen when dealing with different *personas*. Apparently, the flow should be the same, but it's not.

Conference organizers or track hosts can invite some speakers, while others submit their proposals in the Call for Papers. The flows can be independent in the upstream part of the flow (you may want to skip a cumbersome review process for a superstar speaker). Downstream they're probably not (on the conference schedule, you want the same data, regardless of how you got it).



Two parallel flows may require independent models.

Some organizations are well-equipped to think in terms of *roles*: they'll recognize that speakers and keynote speakers are different in the left part of the flow, but they'll have a similar badge during the *registration process*, and they won't be different from regular attendees during lunchtime, when their role would be a simple `mouth to feed`.

Heuristic: look at the humans in the room

This is so obvious that I feel embarrassed to mention, but here we are: the people. Where people are during the exploration is probably giving the simplest and powerful clue about different model distribution.

Experts tend to spend most time hovering around *the areas that they know better*, to provide answers or to *correct wrong stickies*[15] that they see on the paper roll. Alternatively, they comment around *the areas that they care about*, maybe because the current implementation is far from satisfactory.

Different people are a great indicator of **different needs**, which means **different models**.

[15]Nobody can resist this temptation: *somebody is wrong here!* I have to point it out immediately! EventStorming leverages this innate human behavior and turns into a modeling propeller.

The fun part is that this information — where people *are* — will never be documented, but will often be remembered, through some weird spatial memory.

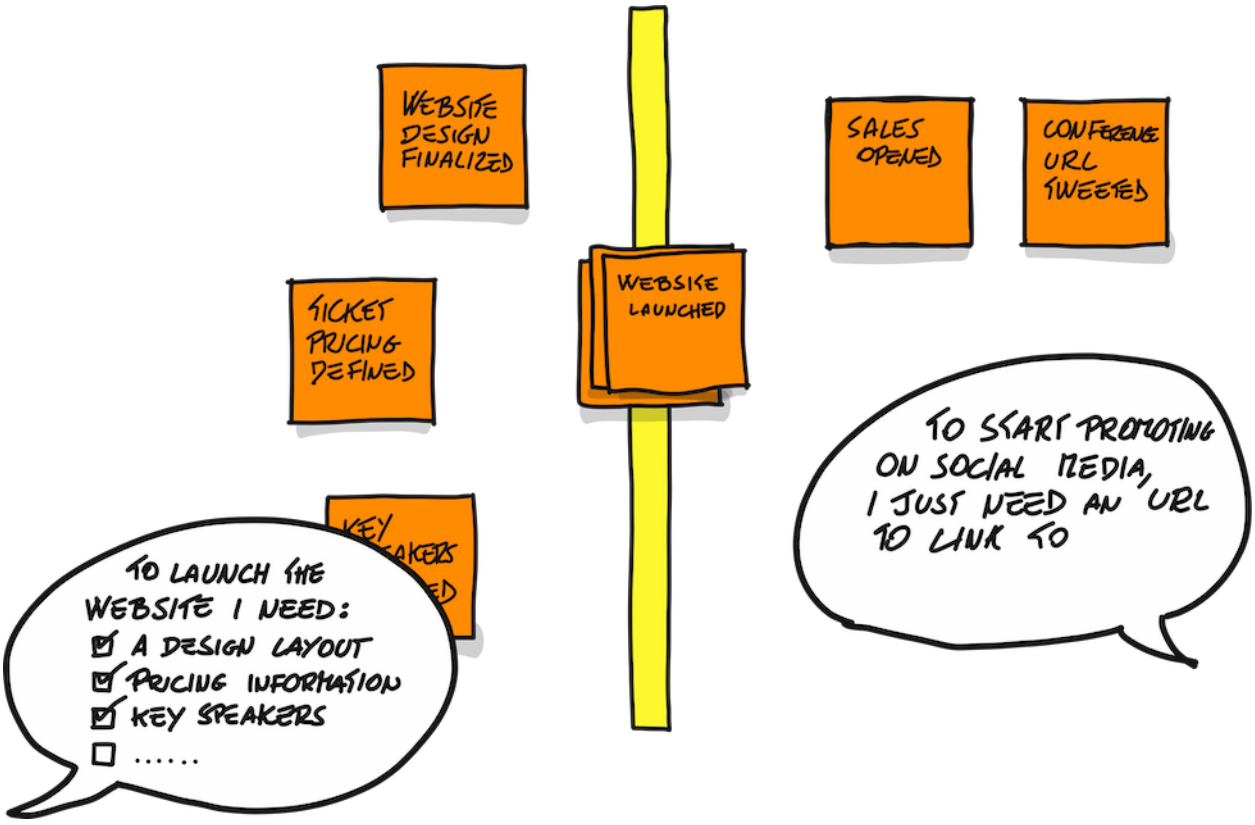
Heuristic: look at the body language

People's body language can be another source of information: not every dissent can be verbal. It's not infrequent to have people from different hierarchy levels to have different views on *apparently the same problem*. Shaking heads, or eyes rolling are a clue of conflicting perspectives that haven't been addressed.

Domain-Driven Design has a fantastic tool for resolving these conflicts: it's not *"we need a model to solve these issues"*, it's *"we need a model to solve your problem* ***and*** *we need a model to solve your boss' problem"*, it would be up to software architects to find the perfect way to interact.

Once again: **different needs** mean **different models**.

It doesn't end here. A typical conversational pattern often happening around pivotal or boundary events is the one in the picture below.



A typical competence clash, the persons on the left usually know all the mechanics involved in a given step, while the ones on the right only care about the outcome.

There is complex knowledge about *everything is needed to complete something* and it's often similar to a task list. On the downstream side of the pivotal event, this complexity should vanish: people

usually don't care about the *how*, but only about the *what*.

In our scenario it may be something like: *"I don't care whether you used WordPress to publish the website, I just need to know whether the URL is reachable or not"* or *"I don't care how did you decide those prices, I just need to know the amount for every ticket type"*.

Heuristic: *listen* to the actual language

This is probably the trickiest tip because language will fool you. The language kept fooling us for decades, and that's one of the reasons why Domain-Driven Design exists.

If you look for central terms like `Talk`, you'll discover that they're used in many different places.

- A `Talk` can be *submitted*, *accepted* or *rejected* in the call for papers.
- A `Talk` can be *scheduled* in a given slot, of a given track.
- A `Talk` can be *assigned* to a given presenter or staff member, to introduce the speaker.
- A `Talk` can be *rated* by attendees.
- A `Talk` can be *filmed* and *recorded*.
- A `Talk` can be *published* on the conference YouTube channel.

…are we sure we're talking about the same `Talk`?

The trick here, is that *nouns* are usually fooling us. People tend to agree on the meaning of *names* by looking at the static data structure of the thing: something like *"A talk has a title."* which is an easy statement to agree with, but doesn't mean we're actually talking about the same thing.

In the list above, we're talking about different models: *selection*, *scheduling*, *staffing*, etc. The thing has probably the same name, and needs to have some data in common between the different models …but *the models are different!*

Looking at *verbs* provides much more consistency around one specific *purpose*.

Putting everything together

Compared to traditional, formal requirements gathering, the amount of information that we can achieve during an EventStorming session is not only superior: it's *massively overwhelming*.

People's behavior and body language can never fit into standard documentation. However, this non-formal information tends to find its way to a good model because *…we've been there!* We've seen people in action around *their problem* and some stupid things like mixing things just because they happen to have the same name, won't happen!

It doesn't require that much discipline, or rules. It simply feels incredibly stupid to mix things that shouldn't, because they don't belong together. They were meters apart on the wall!

I hope the heuristics I just described will help you to sketch your models, but, more importantly, this will give you the chance to understand the deep purpose of your software, and maybe of your organization too, in a compelling call to do the right thing.

In a single sentence, the whole idea is really simple:

Merge the people, split the software.

In retrospective, I still wonder why we wasted all those years doing the opposite.

Emergent Contexts through Refinement — Mathias Verraes

Which Bounded Context owns a particular concept? One way to find out is by evolving your model until everything finds a natural place. All models are wrong, especially the early ones. Let's look at some simple requirements, and explore how we can evolve the model over time. As we learn more about the problem we're solving, we can bring that clarity into new iterations of the model.

The problem

Imagine working on a business application, that deals with sales, accounting, reporting, that sort of thing. The existing software has some serious issues. For example, monetary values are represented as scalars. In many places, values are calculated at a high precision, and then rounded down to 2 decimals, and later used again for high precision calculations. These rounding errors are all over the code. It doesn't make a huge difference on a single amount and a single rounding error, but eventually it could add up, and cost the business millions. The monetary values can represent different currencies, but the financial reporting is always in EUR. It is unclear if the code always correctly converts to EUR when needed, or accidentally adds up amounts from different currencies.

To solve these problems, the first thing to do is to have conversations with the domain experts from sales and accounting. As a result, we can come to a common agreement on some requirements.

Requirements

1. Business needs to support about 10 currencies, possibly more in the future. When business wants to support a new currency, say Japanese Yen, then it is assumed that the developers will add support in the code. There's no need for a UI for adding new currencies.
2. All price calculations need to be done with a precision of 8 decimals. This is a business decision.
3. When showing amounts to users, or passing them along over an API, the software will always stick to the currency's official division. For example, in the case of EUR or USD, that's 2 decimals. In the case of Bitcoin, a precision of 1 satoshi aka BTC 10^{-8}.
4. In some markets, the software will need to do specific compliance reporting.
5. All internal reporting needs to be in EUR, no matter what the original currency was.
6. There will be some legacy and third-party systems, that also publish revenues to the internal reporting tool. Most of them only support the currency's official division, and can't deal with higher precision.

Some programming languages don't deal well with highly precise calculations. There are workarounds which won't be discussed in this article. Just assume that here "number" means a suitable datatype, such as float or Decimal in C# or BigDecimal in Java.

The first step

Are there existing design patterns that can help here? A good pattern to apply when dealing with monetary values is the Value Object pattern from Domain-Driven Design[16]. In fact, a couple of years before Eric Evans published his book, Martin Fowler described an implementation for `Money` in Patterns of Enterprise Application Architecture[17]. The pattern describes an object consisting of two properties, a number for the amount, and another property for the associated currency (which could be a Value Object as well). The `Money` Value Object is immutable, so all operations will return a new instance.

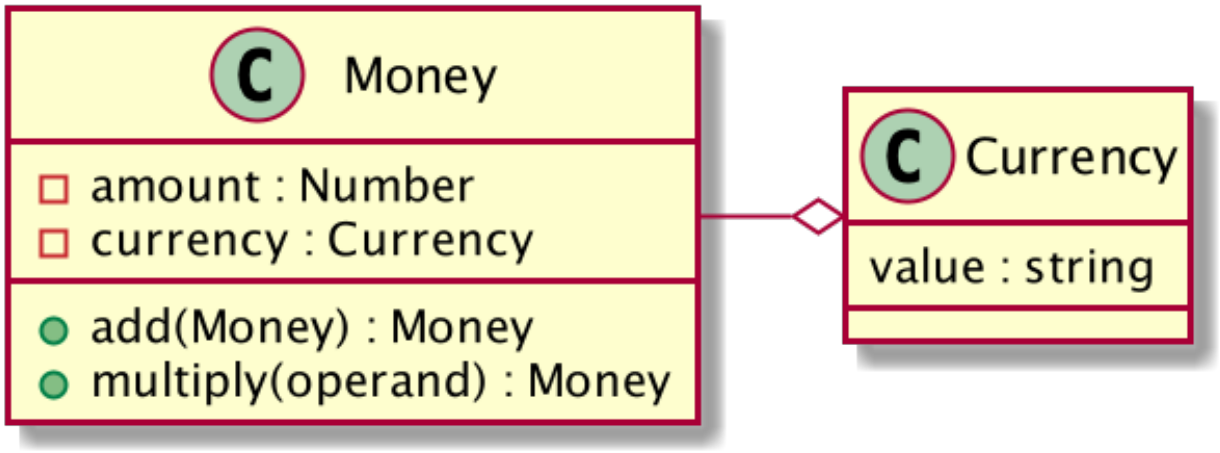


Figure 1

The `Currency` type supports the 10 different currencies that the business is interested in. Enums can be used, but a simple assertion will do if the language doesn't have support for enums. The following constraints can be added:

- The type can't be initialised with a value different from any of the supported 10 currencies.
- It only uses the 3-letter ISO symbols, anything else is an error. That satisfies requirement 1.

`Money`'s constructor rounds the numbers to 8 decimals. This gives it enough precision to deal with any currency. Operations like `Money.add(Money other) : Money` and `Money.multiply(Number operand) : Money` etc can be added to the `Money` class. The operations automatically round to 8 decimals. In addition, there's also a `Money.round() : Money` method that returns a new Money object, rounded to 2 decimals.

[16] http://amzn.to/1CdXXP9
[17] http://amzn.to/1TN7Tq4

```
Money {
    // ...
    round() : Money {
        return new Money(round(this.amount, 2), this.currency)
     }
}
```

Now that we have modeled the types based on the requirements, the next step is to refactor all the places in the old code that do things with money to use the new `Money` object.

Composition

An interesting aspect of Value Objects, is that they are the ultimate composable objects. That helps to make implicit concepts more explicit. For example, we could naively use the money type to represent prices. But what if a price is not so simple? In Europe, prices have a Value Added Tax component. You can now naturally compose a new `Price` Value Object from a `Money` and a `VAT` object, the latter representing a percentage.

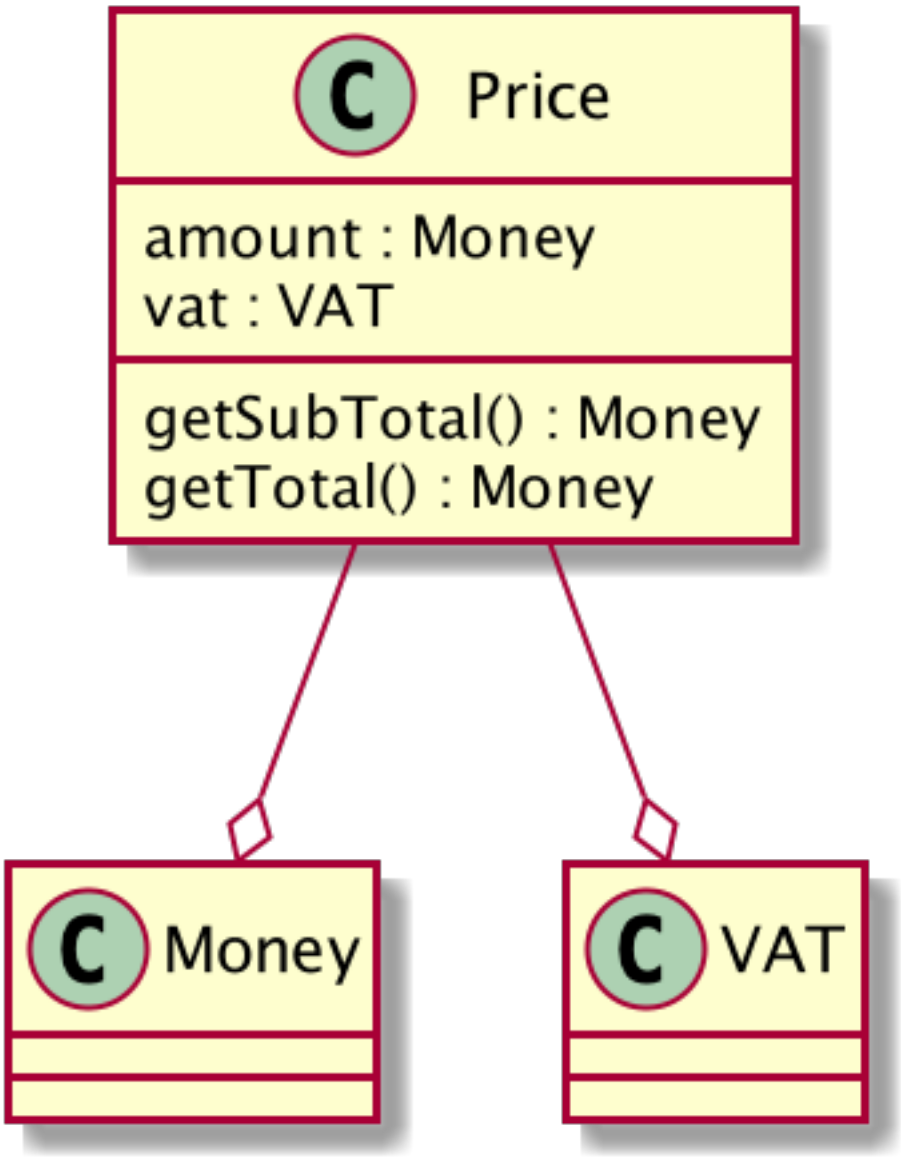
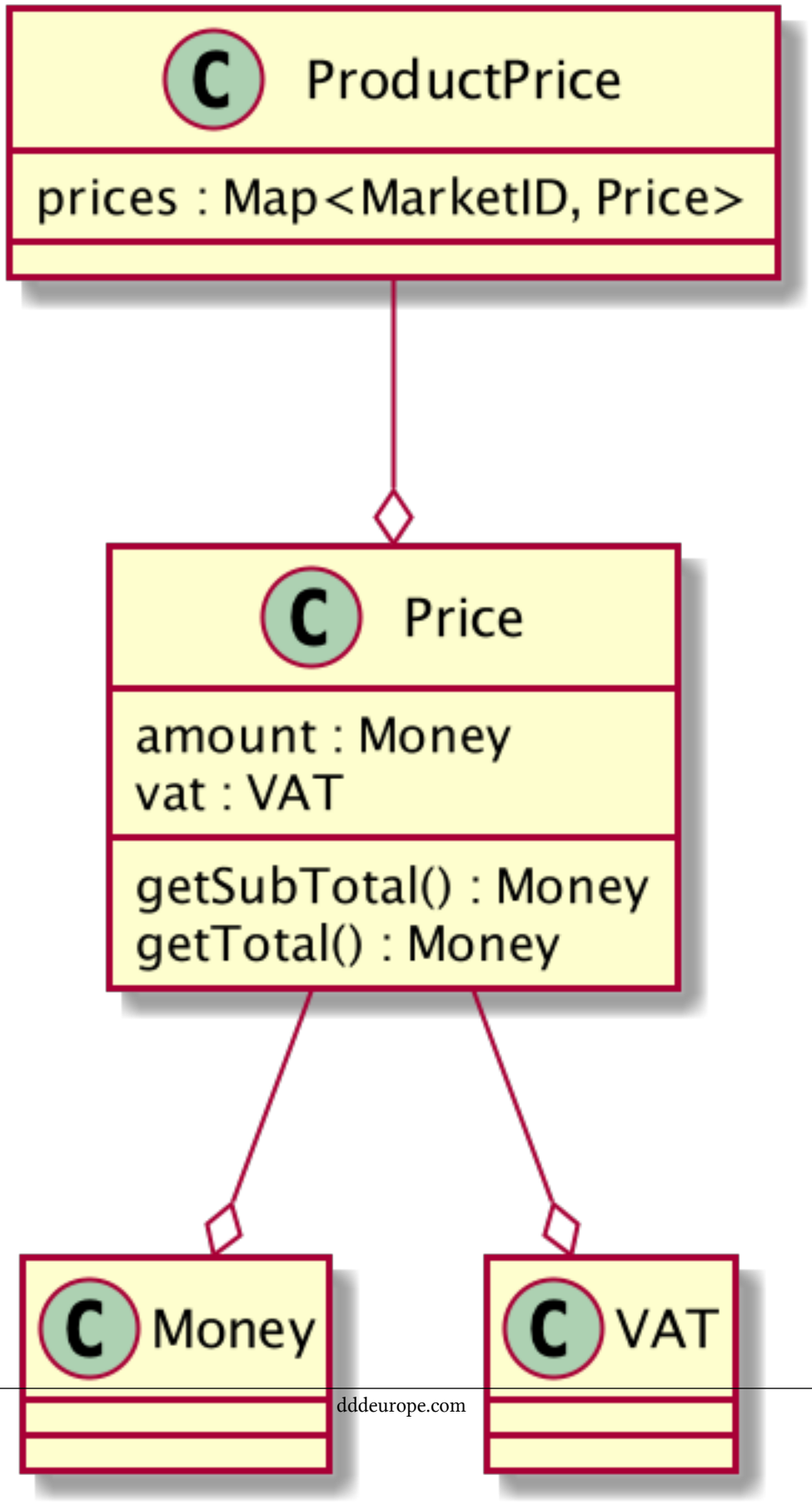


Figure 2

A Product could have different prices in different markets, and we can make this concept explicit in a another Value Object.


ProductPrice
prices : Map<MarketID, Price>
Price
amount : Money
vat : VAT
getSubTotal() : Money
getTotal() : Money
Money
VAT

… and so on. We can push a lot of logic that we'd traditionally put in services, down into these Value Objects.

Value Objects make it easy to build abstractions that can handle lots of complexity, at a low cognitive cost to the developer using it. Finding the abstractions can be hard, but spending the effort here usually impacts code quality and maintainability so much in the long run, that it's very often worth it.

Look for weaknesses

Even if the current model feels intuitively good, there is always room for improvement. Eric often suggests to look for aspects that are awkward or painful to use. These are smells, pointing to opportunities for further refinement.

Looking closely at the model, there are two potential weaknesses:

1. While the scalars in the code have been replaced by the `Money` type, the software is still at risk that money is being rounded and then reused in a higher precision calculation. This problem of rounding errors still has not been fully addressed.
2. The software supports `8` decimal precision only, and the model assumes this is fine. However, when `Money.round()` gets invoked, the model doesn't really deal well with the fact that some currencies don't have two but three decimals by default (such as Bahraini and Kuwaiti Dinar) or more (like Bitcoin), and some have none (like the Japanese Yen).

This is where we should start feeling an itch to stop and rethink our model.

Have a close look at: `Money.round() : Money`

```
a = new Money(1.987654321, eur)
// The constructor rounds it down to 1.98765432
b = a.round()
// round() rounds it up to 1.99 and instantiates a new Money
// Money's constructor rounds it to 1.99000000
```

Technically, most programming languages don't distinguish between `1.99` and `1.99000000`, but logically, there is an important nuance here. `b` is not just any `Money` , **it is a fundamentally different type of money**. The current design doesn't make that distinction, and just mixes up money, whether it was rounded or not.

Make the implicit explicit.

A good heuristic when modelling, is to consider if we can be more explicit in the naming, and tease out subtly different concepts. We can rename `Money` to `PreciseMoney`, and add a new type called `RoundedMoney`. The latter always rounds to whatever the currency's default division is.

The `Money.round()` method now becomes very simple:

```
PreciseMoney {
    round() : RoundedMoney {
        return new RoundedMoney(this.amount, this.currency)
     }
}
```

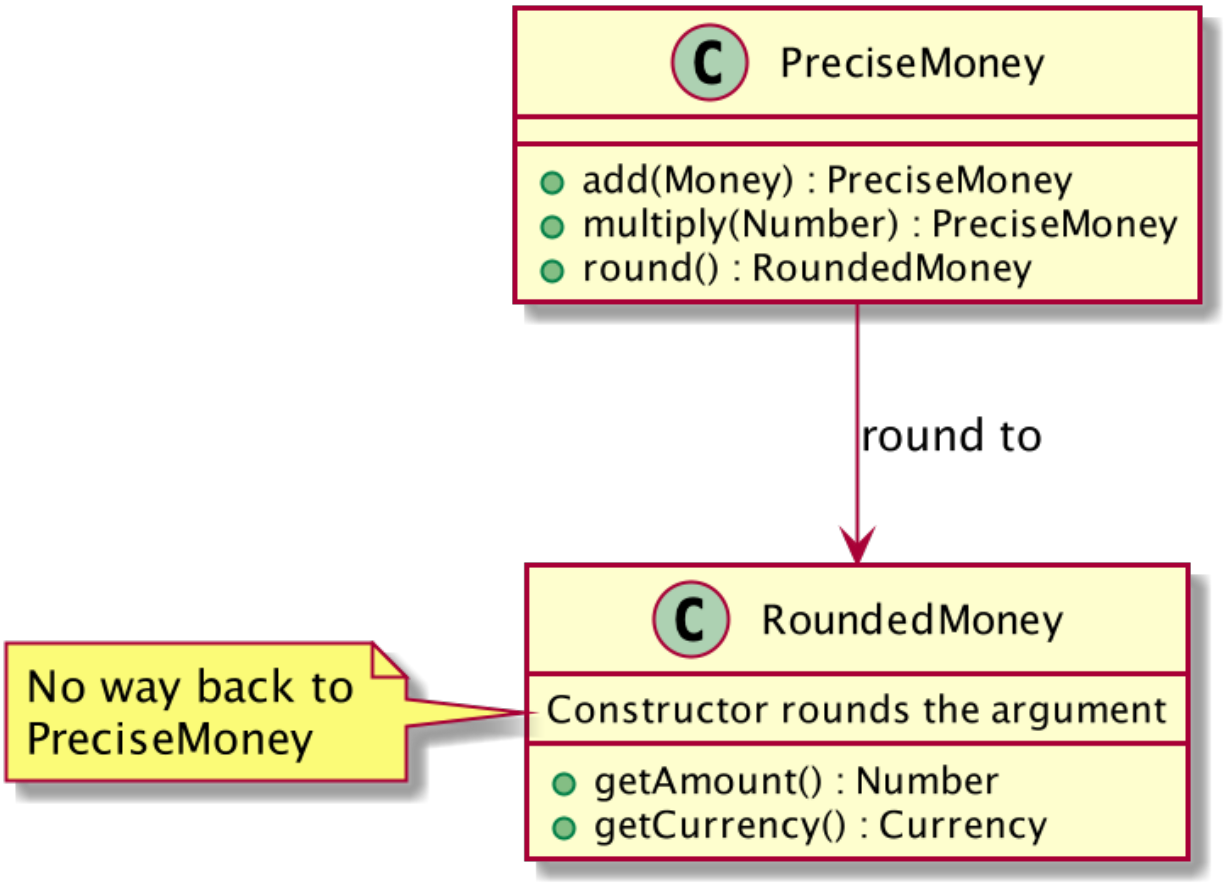


Figure 4

The chief benefit is strong guarantees. We can now typehint against `PreciseMoney` in most of our domain model, and typehint against `RoundedMoney` where we explicitly want or need it.

It's easy to underestimate how valuable this style of granular types can be.

- **It's defensive coding, against a whole category of bugs**. Methods and their callers now have an explicit contract, about what kind of money they are talking about. Good design communicates intent.
- **Contracts beat tests**. Obviously correct code doesn't need tests. If `RoundedMoney` is well-tested, and some client code typehints for that type, we don't need a test to verify that this money is

in fact rounded. This is why proponents of strong static type systems like to talk about *Type Driven Development* as an alternative to *Test Driven Development*: Declare your intent through types, and have the type checker do the work.

- **It communicates to different people working on the code, that we care specifically about the difference**. A developer who is unfamiliar with this problem space, might look for a `Money` type. But the IDE tells them no such type exists, and suggests `RoundedMoney` and `PreciseMoney`. This forces the developer to consider which type to use, and learn something about how the domain deals with precision.
- **It introduces a concept from the domain into the Ubiquitous Language and the model**. Precision and Rounding was fundamental to the domain but clearly ignored in the original model. Co-evolving the language, the models, and the implementation, is central to Domain-Driven Design. These model refinements can have exponential payoffs in the long run.
- This design also **helps to apply the Interface Segregation Principle**. Client code will not depend on an large set of APIs, only on the ones relevant to the type.

Dealing with different precision

We can add a `round()` method to `PreciseMoney`, but we wouldn't add a `toPrecise()` method to `RoundedMoney`. Just like in physics, we can't create more precision out of nothing. In other words, you can cast `PreciseMoney` to `RoundedMoney`, but you can't cast `RoundedMoney` to `PreciseMoney`. It's a one way operation.

There's an elegance to that constraint. Once you round something, the precision is lost forever. The lack of `RoundedMoney.toPrecise()` fits the understanding of the domain.

Common Interfaces

You may have noticed that in the current design, there is no `Money` interface at the top of the object graph. Aren't `PreciseMoney` and `RoundedMoney` both a kind of `Money` or `AbstractMoney`? Don't they share a lot of methods?

If the goal is to try and build a model inspired by the real-world, this would make sense. However, don't judge your models by how well they fit into hierarchical categorisations. A Domain Model is not an taxonomy — in contrast to the OOP books that teach you that a `Cat extends Animal`. Judge your models based on usefulness instead. A top-level `Money` interface adds no value at all; in fact it takes away value.

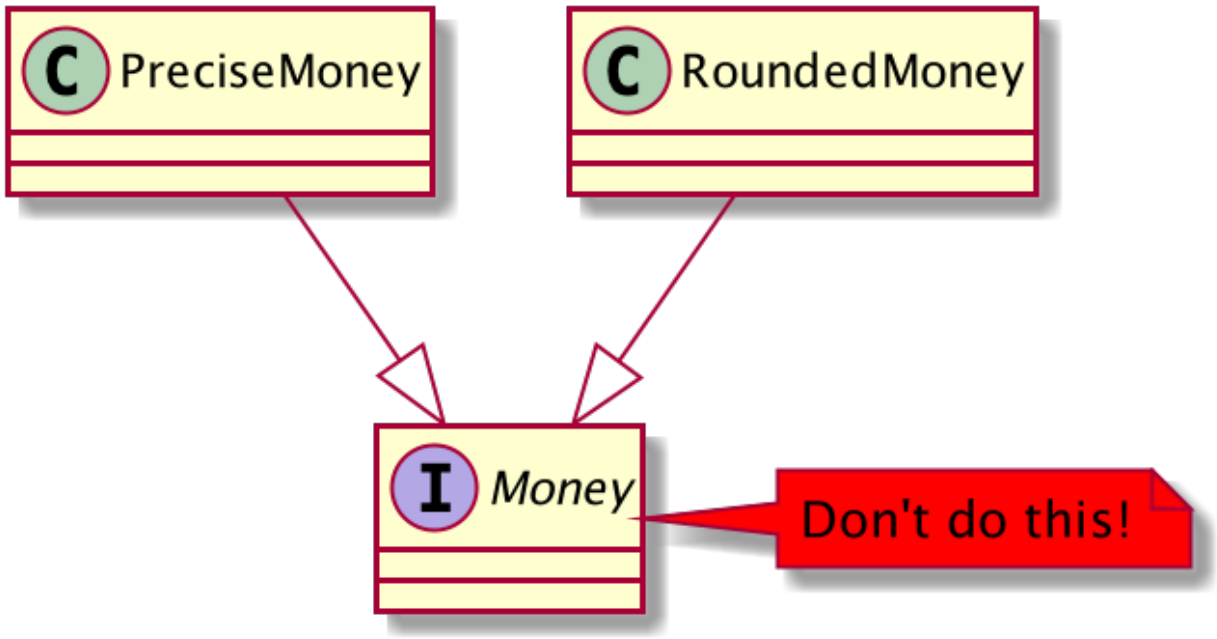


Figure 5

This may be a bit counterintuitive. `PreciseMoney` and `RoundedMoney`, although somewhat related, are fundamentally different types. The model is designed for clarity, for the guarantee that rounded and precise values are not mixed up. By allowing client code the typehint for the generic `Money`, you've taken away that clarity. There's now no way of knowing which `Money` you're getting. All responsibility for passing the correct type is now back in the hands of the caller. The caller could do `money instanceof PreciseMoney`, but that's a serious code smell.

Dealing with Conversions

Converting between different currencies depends on today's exchange rates. The exchange rates probably come from some third party API or a database. To avoid leaking these technical details into the model, we can have an `CurrencyService` interface, with a `convert` method. It takes a `PreciseMoney` and a target `Currency`, and does the conversion.

```
1  interface CurrencyService {
2     convert(PreciseMoney source, Currency target) : PreciseMoney
3  }
```

The `CurrencyService` implementations might deal with concerns such as caching today's rates, to avoid unnecessary traffic on each call.

If fetching + caching + converting sounds like a lot of responsibility for one service, that's because it is. Service classes like are `CurrencyService` fundamentally procedural code wrapped in an object.

Even though leakage is reduced, thanks to the `CurrencyService` interface, the rest of the code still needs to depend on this interface. This makes testing harder, as all those tests will need to mock `CurrencyService`. And finally, all implementations of `CurrencyService` will need to duplicate the actual conversion, or somehow share that logic. A heuristic that helps, is to figure out if we can somehow isolate the actual domain logic code from the part of the code that is just trying to prepare the data for the domain logic to be applied. For example, differentiate the domain logic (conversion) from the infrastructure code (fetching, caching).

There's a missing concept here. Instead of having the `CurrencyService` do the conversion, we can make it return a `ConversionRate` instead. This is a Value Object that represents a source `Currency`, a target `Currency`, and a factor (a float). Value Objects attract behaviour, and in this case, it's the `ConversionRate` object that becomes the natural place for doing the actual calculation.

```
interface CurrencyService {
    getRate(Currency source, Currency target) :: ConversionRate
}
ConversionRate {
    - sourceCurrency : Currency
    - targetCurrency : Currency
    - factor : Float
    convert(PreciseMoney source) : PreciseMoney
}
```

The `convert` method makes sure that we never accidentally convert USD to EUR using the rate that was actually meant to convert GBP to JPY. It can throw an exception if the arguments have the wrong currency.

The other cool thing here is that there's no need to pass around `CurrencyService` interface. Instead, we pass around the much smaller, simpler, `ConversionRate` objects. They are, once again, more composable. Already the possibilities for reuse become obvious: for example, a transaction log can store a copy of the `ConversionRate` instance that was used for a conversion, so we get accountability.

Simpler Elements

An `CurrencyService` is now something that represents the collection of `ConversionRate` objects, and provides access (and filters) on that collection. Sounds familiar? This is really just the *Repository* pattern! Repositories are not just for Entities or Aggregates, but for all domain objects, including Value Objects.

We can now rename `CurrencyService` to `ConversionRateRepository`. It has the benefit of being more explicit, and more narrowly defined. Having the pattern name `Repository` in the class name is a bit of an annoyance. As Repositories in DDD really represent collections of domain objects, calling it `ConversionRates` is a good alternative. But we should take the opportunity to look for terms in

the Ubiquitous Language. A place where you can get today's conversion rates, is a foreign exchange, so we can rename `ConversionRateRepository` to `ForeignExchange` (or `Forex`). Rather than put the word Repository in the classname, we can annotate `ForeignExchange` as a @Repository, or have it implement a marker interface called `Repository`.

The original procedural `CurrencyService` is now split into the two simpler patterns, Repository and Value Object. Notice how we have at no point removed any essential complexity, and yet each element, each object, is very simple in its own right. To understand this code, the only pattern knowledge a junior developer would need, is in a few pages of chapters 5 & 6 of Domain-Driven Design[18].

Currency Type Safety

There are still have some issues left. Remember that some currencies have a division of 1/00, 1/1000, or 1/100000000. `RoundedMoney` needs to support this, and in this case, for 10 different currencies. The constructor can start to look somewhat ugly:

```
switch(currency)
    case EUR: this.amount = round(amount, 2)
    case USD: this.amount = round(amount, 2)
    case BTC: this.amount = round(amount, 8)
etc
```

Also, every time business needs to support a new currency, new code needs to get added here, and possibly in other places in the system. While not a serious issue, it's not exactly ideal either. A switch statement (or a key/value pairs) can be a smell for missing types.

One way to achieve this is to make `PreciseMoney` and `RoundedMoney` abstracts or interfaces, and factor the variation out into subtypes for each currency.

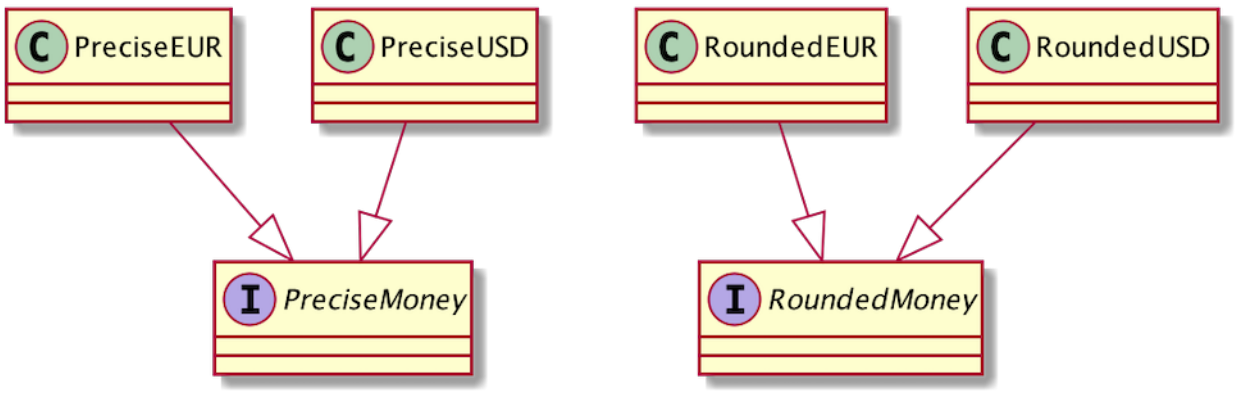


Figure 6

[18] http://amzn.to/1LrjmZF

Each of the `PreciseEUR`, `PreciseBTC`, `RoundedEUR`, `RoundedBTC` etc classes have local knowledge about how they go about their business, such as the rounding switch example above.

```
RoundedEUR {
    RoundedEUR (amount) {
        this.amount = round(amount, 2)
    }
}
```

Again, we can put the type system to work here. Remember the requirement that the reporting needs to be in EUR? We can now typehint for that, making it impossible to pass any other currency into our reporting. Similarly, the different compliance reporting strategies for different markets can each be limited to the currencies they support.

So, when should you *not* want to lift values to types?

Let's say you're not dealing with 5 or 10 values, but there is an infinite or very large set of potential values. Or when you're expecting each of these implementations to change often. In this case, having to support 10 currencies is somewhat on the edge. But again, it's very unlikely that you're having to add support for all 180 currencies. You're probably going to only support the 10 or 15 most relevant ones.

Minimalist Interfaces

At this point, we can start seeing how concepts from our requirements, and concepts from our refined Money model, start clustering. We can start breaking up this model, moving and copying parts to different Bounded Contexts. The name Money fooled us: Money means different things to different Contexts.

A benefit of having lots of small classes, is that we can get rid of a lot of code. Perhaps the Sales Bounded Context deals with the 10 `PreciseXYZ` types, but the Internal Reporting Bounded Context only needs `RoundedEUR`. That means there's no need to support `RoundedUSD` etc, as there's no need for it. This also implies that we don't need `round()` methods on any of the `PreciseXYZ` classes, apart from EUR. Less code means less boilerplate, less bugs, less tests, and less maintenance.

Not supporting a way back from `RoundedEUR` to `PreciseEUR` is another example of a minimalist interface. Don't build behaviours that you don't need or want to discourage.

Single Responsibility

Another benefit of these small, ultra-single-purpose classes, is that they very rarely need to change, and only when something in the domain changes. Currencies are very stable concepts, and our model mimics this slow rate of change. A good design allows you to easily add or remove elements, or change the composition of the elements, but rarely requires you to actually change existing code. This in turn leads to fewer bugs and less work and more maintainable code.

Ledger

One advantage of expressing our code in terms of rich, well-adapted domain models, is that sometimes this can lead to finding opportunities for other features. It also makes the implementation of those features very easy.

In this case, a goal of this modelling exercise was to solve the precision problem. Every time a precise value is rounded, what happens to the fractions of cents that the business gains or loses? Is this important to the domain?

If so, we can keep a separate ledger for rounding. Every time a fraction of the value is gained or lost by rounding, we record it in the ledger. When the fractions add up to more than a cent, we can add it to the next payment.

```
1 // rounding now separates the rounded part from the leftover fraction
2 PreciseMoney.round() : (RoundedMoney, PreciseMoney)
3 Ledger.add(PreciseMoney) : void
```

You might not have to deal with this in most domains, but when you have high volumes of small transactions, it could make a significant difference. Consider this a pattern for your toolbox.

The point here is that our model is easy to extend. In the original code, where rounded and precise amounts were not clear, keeping track of lost fractions of cents would have been a nightmare. Because the change in requirements is reflected in the types, we can rely on the type checker to find all places in the code that need to be adapted.

Refining Bounded Contexts

Different capabilities of the system have different requirements. This is what Bounded Contexts are good for: they allow us to reason about the system as a number of cooperating models, as opposed to one unified model. Perhaps the Product Catalog needs a really simple model for money, because it doesn't really do anything other than displaying prices. Sales may need precise price calculations. Our internal Reporting might be fine with reporting with rounded numbers, or even with reporting with no decimals. Company financials are sometimes expressed in thousands or millions. The Compliance Reporting again has different needs.

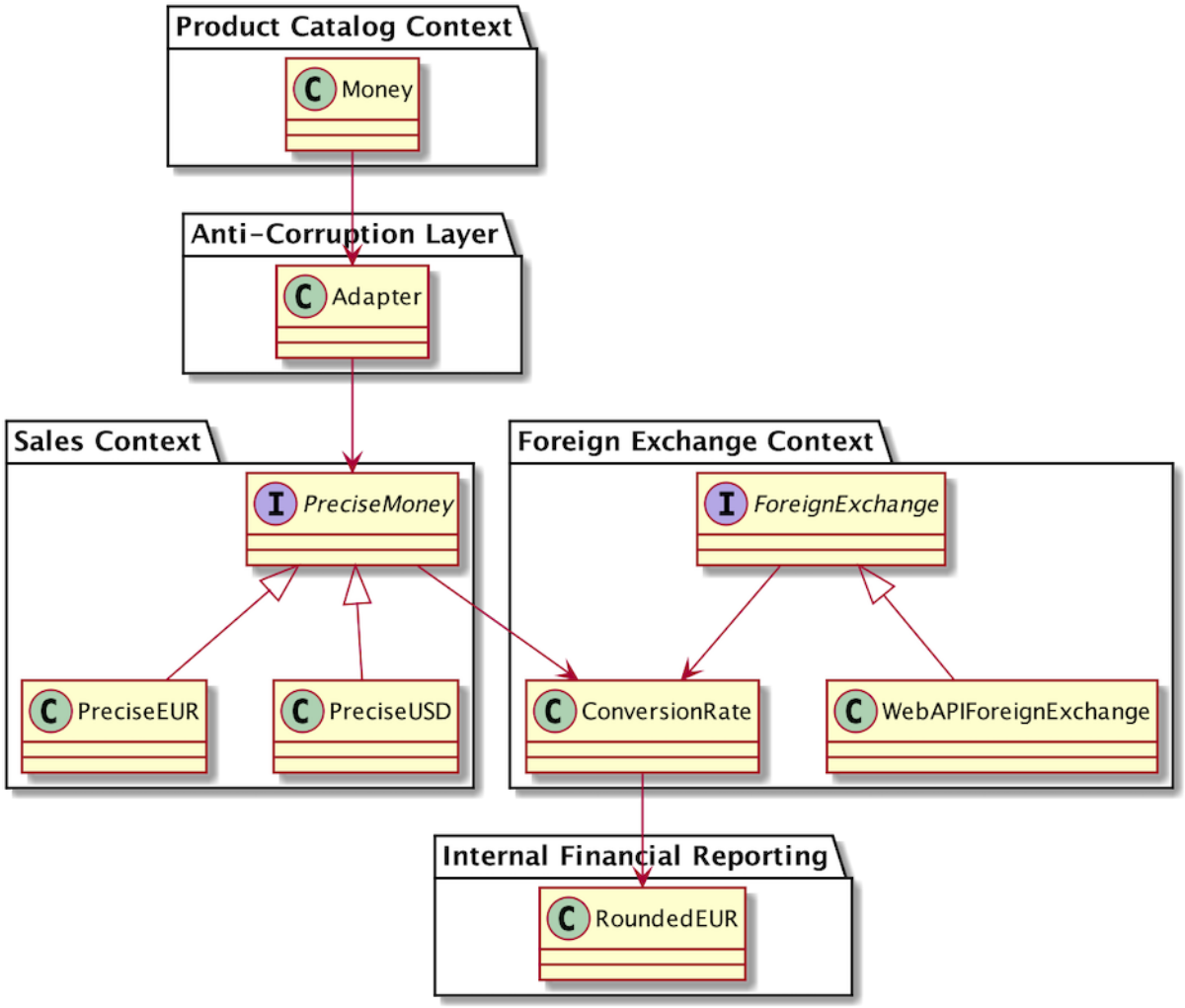


Figure 7

Cleaning up the language:

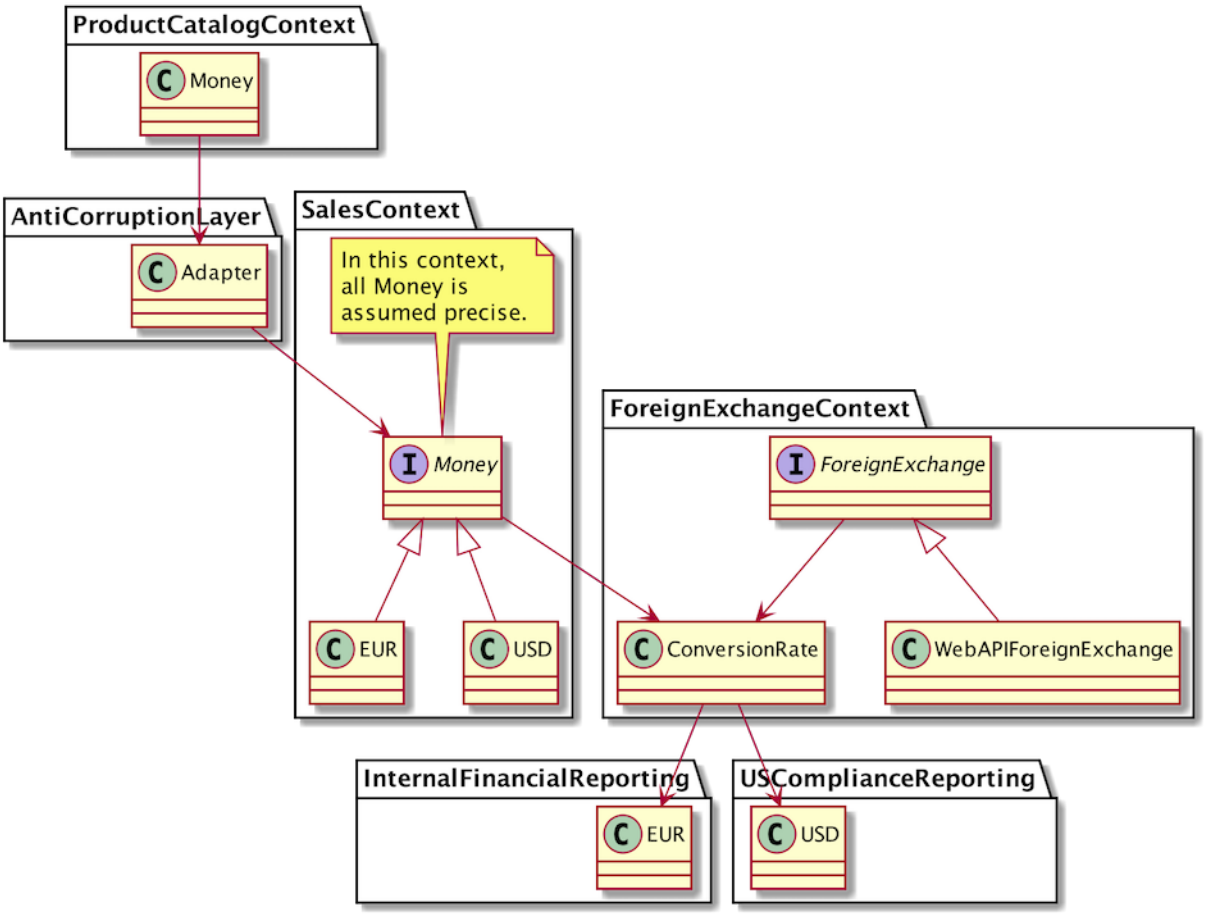


Figure 8

If the Sales context always deals with high precision, does it then make sense to call the type `PreciseMoney`? Why not just `Money`? It should then be clearly understood that in the Sales Ubiquitous Language, there is only one `Money` and it doesn't tolerate rounding. In the Reporting Context, money is always rounded and in EUR. Again the type doesn't have to be `RoundedMoney` or `RoundedEUR`, it can just be `Money`.

Every Bounded Context now gets its own domain models for Money. Some are simple, some have more complexity. Some have more features, others have less. We've already written all that code, we're just separating it into the Contexts where we need it. Each Bounded Context has a small, unique model for Money, highly adapted to its specific needs. A new team member working on a Bounded Context can now learn this model quickly, because it's unburdened by the needs of other Contexts. And we can rest assured that they won't make rounding errors or swap EUR and USD. Different teams don't need to coordinate on the features of Money, and don't need to worry about breaking changes.

In some environments, a single Generic Subdomain for money would have been perfectly fine. In others, we want highly specialized models. Determining which is which is your job as a domain modeller.

We didn't do upfront analysis of what goes into which Bounded Context. If we had tried that, early on in the project, we would have naively built a shared Money library, assuming that the concept of Money is universal between all Contexts. Instead, we kept on refining our model based on our needs and new insights, and saw how these explicit concepts naturally found a place. Cultivate a healthy obsession with language, and keep refactoring towards deeper insight, one step at a time. All good design is redesign.

The Captain of the Night Watch — Indu Alagarsamy

This article is based on a conversation that ensued during a midnight stroll in the streets of Amsterdam with Eric Evans, Paul Rayner, and Mathias Verraes.

There I was, standing in front of a masterpiece, thanking my lucky stars of the incredible opportunity that I had just been given to be there, to be present in that moment. I was in front of Rembrandt's Night Watch, standing in complete awe. The captain was striking and regal. The light flowed into the room, as if the heavens were trying to illustrate his authority. He was giving orders to the people around him. There was a sense of life and belonging, a canvas that was filled with so much emotion all captured so beautifully. I was in the midst of it all, facing the captain and for a brief moment, it almost felt as if he was walking straight toward me.

Ever since I first saw the windmills of Zaanse Schans in a travel book, I was fascinated and wanted to visit Amsterdam someday. So I was beyond thrilled when Mathias invited me to speak at DDD Europe 2018, and the venue was Amsterdam! The universe had granted my wish. Not wanting to waste a single minute, I started my exploration of the city with the Rijksmuseum. When I arrived at the museum, I realized that I had only about an hour, so I asked the museum attendant what's the one thing I shouldn't miss at the museum. Non-judgmentally she said, "Rembrandt's Night Watch." And that's how I first met the captain on a complete coincidence.

Fast-forward to the conference. Despite my jet lag, my talk at DDD Europe went fabulously well, and at the conference closing, I was invited to go to dinner and hang out with the DDD greats: Eric Evans, Paul Rayner, and Mathias Verraes. I felt incredibly lucky. At dinner, we discussed Bollywood movies, and I was pleasantly surprised listening to Eric talk about "Lagaan" and explain the storyline to Paul and Mathias.

As we were making our way back to our hotel we found ourselves in front of the Rembrandt statue at the city square. Rembrandt was standing on a pedestal and right below him, it was as though his painting had come alive at the stroke of midnight. All the characters that Rembrandt had meticulously detailed out in his Night Watch were transformed into bronze statues and brought to life in 3D. There were 22 statues including one for the dog, all standing in the same pose created by Rembrandt, frozen in time for the world to see. And there was the captain again. It felt like deja vu standing face to face with him, this time in 3D.

What does this have to do with Bounded Contexts?

We took a selfie in front of the statues. But then Eric suggested that it's not every day you get to see what's behind the Night Watch, something you don't actually get to see in the actual painting

and that we should be taking pictures from the other side. This brought up the question of different perspectives. Because the statues were 3D, you had a different view, a perspective from behind the figures as well. As we were taking pictures, being a nerd, I had a serious question. I always have a tendency to try and relate software with actual real life things and that helps me understand the bigger picture. So the question that was burning in me this time, at the midnight hour was: would the front and back of the statues make different bounded contexts since it was essentially a different perspective of the same thing? Do different perspectives make different bounded contexts? It was as if the captain was trying to show me something important, but I had to try hard to grasp it. Luckily for me, I was with the right company, so I asked Eric. Eric with his calm and logical way of explaining things answered me without a second thought that the painting and the sculpture would be the two different bounded contexts and not the front and the back of the captain.

What makes the painting and the sculpture a different context?

I let that sink in for a moment. So the statue and the painting were the different contexts. Then like a puzzle piece falling into place, it started to make sense to me. The biggest reason is because the rules around these contexts were completely different. The constraints of making art on a canvas and sculpting a bronze statue are entirely different. While the captain might appear differently from the front and the back, the constraints on the statue were the same, regardless of which angle you looked at him.

I remembered the evening at the museum where I stood in front of the Night Watch. Light was streaming down on the captain as if to highlight his authority. But at the square, the captain was just one among the 22 individual sculptures. Unless you know about Rembrandt's Night Watch, you couldn't tell who was the man with all the authority. At the square there was nothing special about the captain, and definitely no special lighting. The constraints on the light were different in both the painting context and the sculpture context.


Comparison of the Captain in the painting vs sculpture

When Rembrandt was painting, he didn't have to worry about how these characters were going to look from behind. 2D was perfectly sufficient. However, when the Russian artists created the sculptures they had to deal with a whole new set of constraints that pertained to creating statues out of bronze in 3D.

In Rembrandt's art, the level of detail for each character was different. In the painting, the details on the captain were stunning from the wrinkles on his boots to the stitching on his vest and then some of the other characters were vague, for example the dog. The sculptures however were all equally well defined. There was no difference in the amount of detail in the captain's statue vs. the dog's statue.

Comparison of the dog in the painting vs sculpture

Also, in the painting there is a character hidden in the group, that's believed to be Rembrandt himself. Whereas at the square, Rembrandt was in plain sight towering over all of his creation from his place on the pedestal. So on the surface, while the art hanging at Rijksmuseum and the sculpture at the city square both represent the Night Watch, they are bound by different constraints, have different goals, and therefore different focus and detail. That makes them different bounded contexts.

How does this help with software design?

In software design, we might have to model the concept of a "Product." However what the Product is to the "Sales" context is different from that of an "Inventory" context and that of a "Shipping" context. In the Sales context, a product brings in revenue to the company. In the Inventory context, it's a thing that you have (or don't have) in some quantity. In the Shipping context, it's a thing that has some weight associated with it, which gets sent from, say, Amsterdam to California. On the surface, just like the Night Watch, it's the same. However, depending on the context, the rules are different.

We have been taught early to look for nouns to help determine the objects. , We've been so well trained to find the nouns, it's hard to think differently. In that mindset, a product is a noun; therefore it gets its own model. Things like name, description, price, how much of it you have in stock, what it would take to ship it, etc., tend to get associated with this product object model. OO always stressed behavior and data hiding and data encapsulation. But it's too easy to turn off that behavior bit part of it and just model based on pure data. But here's the thing: would you require transactional integrity when trying to update the description of the product, compared to where you were trying to ship it? Are there any business rules to say, "All products that start with X in the name cannot be shipped to California?". If you don't have any such rules, then having the product's name and the shipping rules in the same transactional boundary doesn't make much sense.

Here is where Eric's Domain-Driven Design forces you to think of individual models, as opposed to the "one universal model to rule them all". Placing more emphasis on what processes happen in the business, and focusing more on the pain points, and identifying the constraints, helps to model the context more accurately, rather than just trying to model the data in a entity driven fashion. Properties like Description and Price can be separated from the context if we apply the rule that each context must be isolated, internally consistent, and unambiguous. And most importantly that the context can have the same representation of a concept, just that different rules govern it. Just like Rembrandt's Night Watch on the canvas and the sculptures in the city square.

As all nights do, this one came to an end too. I felt like I had come a full circle. From that first moment when I was in awe of the Captain at Rijksmuseum to seeing him again at the square, both Rembrandt and Eric in their own ways, showed me the nuances of the bounded context. While I had to part my ways with the captain, I walked away with a new learning and appreciation for Bounded Contexts.

Group Selfie

Traces, Tracks, Trails, and Paths: An Exploration of How We Approach Software Design — Rebecca Wirfs-Brock

This work is based on an essay I presented and workshopped at the PLoP (Pattern Languages of Programs) 2018 Conference held October 24-26, 2018 in Portland, Oregon, USA

Introduction

If I were to be brutally honest about the nature of software design, I would give up on any notion of certainty. The more I know about software and the world it is part of, the more cautious I become about making absolute statements about either. Software design is full of unexpected complexities and continual surprises. I cannot predict which contextual details will suddenly become important. Small details can loom large and undo even the best design intentions.

Because I acknowledge this uncertainty, I seek out other designers' stories. I want to learn about the ugly, confusing aspects of design that are rarely written about. I want to incorporate others' insights into my growing understanding of the nature of software design. I want to learn what heuristics they use to solve their design problems and see where they clash with or complement my own.

As a designer I often encounter conflicting goals, dynamically changing context, and shifting degrees of certainty about those heuristics I know and cherish. Once in a while this makes me pause to reflect and readjust my thinking. But more often, I quickly take stock of the situation and move on, perhaps only tweaking my design a little, without much exploration or thought. I don't spend much time consciously rethinking and rearranging my worldview.

I'm hoping to change that just a little by giving myself some space and time to reflect on how I approach design and share some ways collectively we as designers might grow, alter, articulate, and better share our heuristics. There is much to learn about design from the stories we tell and from the questions we ask of each other.

Background

A software designer's personal toolkit likely includes an awareness of some hardcore technical design patterns (and how to shape and adapt and refine them). It also includes heuristics for how

to approach the current task at hand. Our heuristics have been imparted to us through code and conversations, as much as anything. While we may read others' design advice—be it from patterns or stack overflow replies, the heuristics we've personally discovered on our own design journey may be even more important.

In *Discussion of the Method*, Billy Vaughn Koen defines a heuristic as, "anything that provides a plausible aid or direction in the solution of a problem but is in the final analysis unjustified, incapable of justification, and potentially fallible." If you desire to create or change a system (whether social, political, physical, software, or otherwise), opting for what you consider to be the best available heuristics to apply as you balance conflicting or poorly understood criteria for success, then you are solving an engineering problem. Rarely are such problems well defined. Instead, we problem solvers determine what the actual problem is based on diffuse, changing requirements. And to solve that problem, we successively apply heuristics based on our imperfect knowledge of both the current situation as well as the outcome of taking any specific action. Heuristics offer plausible approaches, not infallible ones.

When we software designers choose an approach to solve a current problem, most of the time we are satisficing—finding a satisfactory approach, not actively judging what's best or optimal. If a heuristic seems to fit the situation, I try it. Given what I know, what I believe to be salient at the moment, what I intuit, what I value, and what constraints I have, I choose what I think are reasonable heuristics (at whatever granularity they are). There is no guarantee that doing so actually moves me closer to my design goal. Consequently I need to check my emerging solution for flaws or weaknesses. If I spot any, I take corrective action. Sometimes I backtrack a long way, unwinding what I've already done in order to try out an alternative design approach. More often than not, I only slightly backtrack, having already committed myself to a path that I want to follow. In that case I'm not willing to invest in finding a totally new approach. And sometimes, even though things don't seem to be working out, I plow ahead, although I feel uneasy, hoping I'll be on firmer footing soon. I never proceed in a straight line from problem understanding to solution design in a series of even steps. Instead, I move haltingly forward to a more nuanced understanding of what aspects of my emerging solution are important.

Most of the time, I work on autopilot. I make many decisions and take many design actions, using heuristics at whatever level I need. These heuristics have been deeply embedded into my design gestalt. I apply them without any conscious thought. Only when I bump up against a design challenge where I don't know what to do next—when there is some tension or nagging uncertainty or unfamiliar territory—do I actively take a step back from what I'm doing to look outside of myself for others' wisdom. It is when I pop out of this "unconscious action" mode to actively search for a design heuristic that I want to be able to quickly assess the utility of any I might find.

I assert that a well-written pattern is a particularly nicely packaged form of heuristic. Patterns are particularly useful as they are drawn from direct experience and include handy information for the discerning designer—most notably the context where the pattern is useful as well as tradeoffs and consequences of applying it.

Although I like patterns, the vast majority of software design heuristics have not been written in pattern form. Nor do I expect them to be. Not every useful heuristic is a pattern. I seek out those

other heuristics, too. I am on the lookout for useful heuristics wherever I am engaged in designing or learning about software design (for example, when thinking about how to solve a current problem that is unfamiliar, when reading code, reading blogs, when playing with a new framework, when searching for online advice and recommendations, when attending conference talks, talking with friends, going to meetups, …). I keep adding to my bag of tricks. I tweak and refine heuristics through experience. Rearranging and growing my heuristics toolkit is ongoing and not in anyway systematic.

Metaphors for understanding the certainty and utility of different software heuristics

Could I be a better software designer if I made finer distinctions between heuristics? There are those I know deeply and have learned from others. There are those I discovered on my own. There are heuristics I know intimately—however I came to know them—that I have lovingly polished through experience. And there are those shiny new heuristics I hear or read about.

So what are some ways to understand the soundness and utility of heuristics we find? Robert Moor, in his book, *On Trails*, suggests that we untangle the various meanings and distinctions between trails, traces, tracks, ways, roads, and paths in order to understand how trails came to be and continue to evolve.

> "The words we English speakers use to describe lines of movement—trails, traces, tracks, ways, roads, paths—have grown entangled over the years…But to better understand how trails function it helps to momentarily tease them apart. The connotations of trail and path, for example, differ slightly…the key difference between a trail and a path is directional: paths extend forward, whereas trails extend backward. (The importance of this distinction becomes paramount when you consider the prospect of lying down in the path of a charging elephant versus lying down in its trail). Paths are perceived as being more civilized in part because of their resemblance to other urban architectural projects: They are lines projected forward in space by the intellect and constructed with those noble appendages, the hands. By contrast, trails tend to form in reverse, messily, from the passage of dirty feet." —Robert Moor, *On Trails: An Exploration*

Are published software design patterns more like paths or trails? How certain and civilized and planned are these patterns?

I see a resemblance between paths and published pattern collections. Published patterns collections are neatly laid out, organized, and explained. They typically include some sort of map, suggesting connections and arcs of expected usage. They appear systematically arranged. While individual patterns may have mined from their authors' messy design experiences, the way they are presented hides any of that uncertainty. Those authors seem to know their stuff!

Recently I've learned that some pattern authors were not so certain as their writing suggests. Ralph Johnson, in his Sugarloaf PLoP 2014 keynote[19] said that when they wrote Design Patterns, he and

[19] https://youtu.be/ALxQdnOdYXQ

his co-authors found the creational, behavioral, and structural categories for their pattern collection rather dubious. They went ahead with them anyways, for lack of any better organizing scheme. In his keynote Johnson proposed a better way to categorize the GoF patterns (core, creational, and peripheral), stating that some patterns were definitely less useful, or peripheral than others

Likewise, Eric Evans in several talks suggests that the most important patterns in his collection were the Strategic Patterns. If you look at how the patterns in his book are laid out (see Figure 1) there are really two groupings or patterns collections—those concerned with design details for object designs (e.g. Tactical Design Patterns) and those for organizing and understanding the domains in complex software systems (Strategic Design Patterns). Evans believes that while the Tactical Patterns are useful for object-oriented programming, they aren't nearly as important as the Strategic Patterns. He regrets that the Strategic Patterns were in the latter part of his lengthy book, as some readers never get that far. He also points out that a missing pattern, Domain Events, which was only hinted at in his book, has become increasingly important, especially with the increased use of CQRS (Command-Query-Response Segregation) and Event-Sourced architectures to implement Domain-Driven Design models.

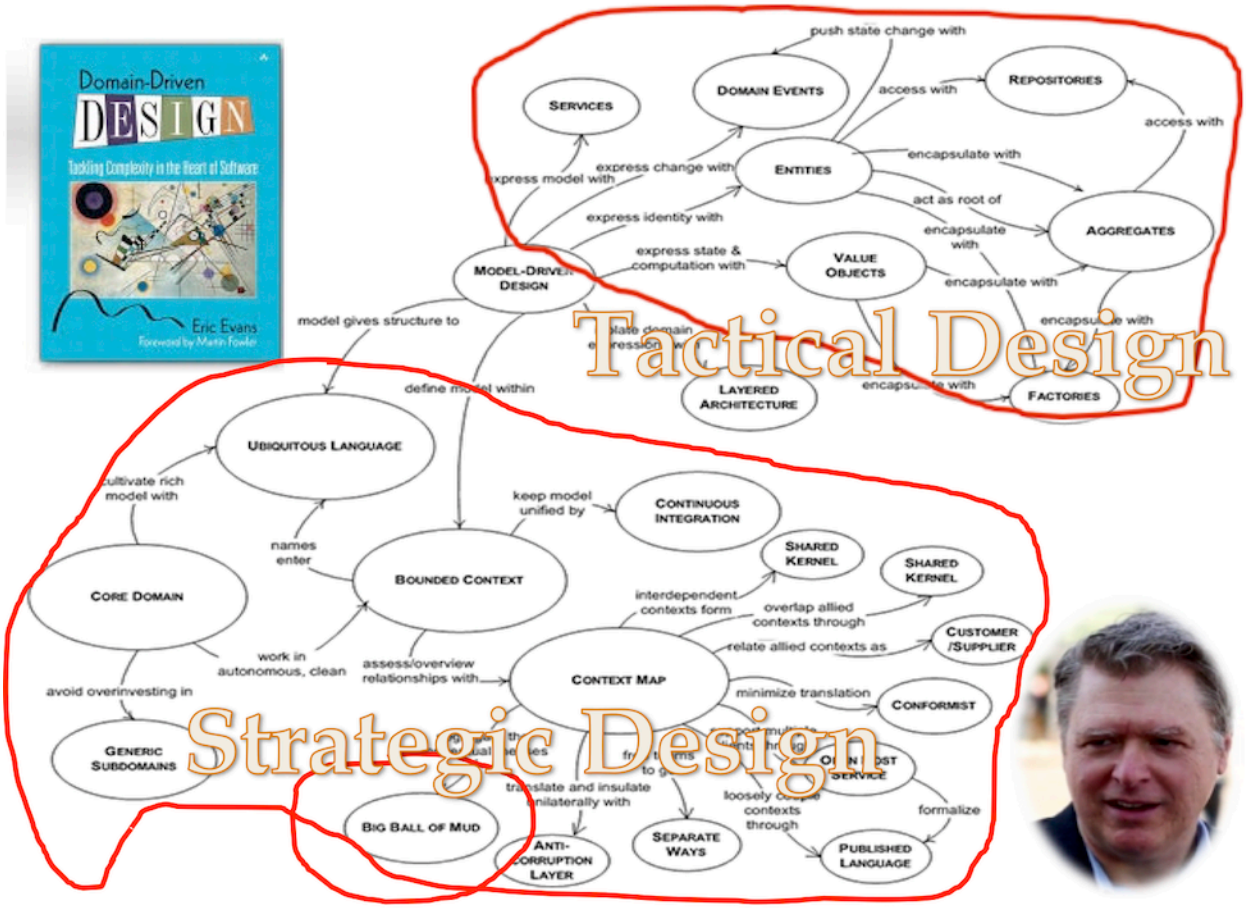


Figure 1. The Domain-Driven Design Patterns are really two collections in one book: Strategic and Tactical Design Patterns

In hindsight, the presentation of these pattern collections seems more tentatively than carefully planned. Had the authors taken time to study how others actually used their patterns, would they have designed better pathways? Or is this something they can see only when looking back on their work?

Perhaps they were really blazing trails instead of constructing pathways.

We can also draw useful analogies between patterns collections and trails. Trails aren't planned and built; they emerge over time. What exactly is it that makes a trail a trail? Richard Irving Dodge, in his 1876 book *Plains of the Great West*, drawing from his experience as a tracker, defined a trail as a string of "sign" that can be reliably followed. "Sign" refers to the various marks left behind by an animal in its passing—scat, broken branches, spoor, etc. A track is evidence; a mark or a series of marks or "sign" that something that has passed through. A track only becomes a trail when a series of "sign" can be followed. Sign, according to Moor, can be physical, chemical, electronic, or theoretical. An animal might leave "sign" but unless it can be tracked reliably, a series of "sign" doesn't automatically make it a trail.

Trails are trails because they can be trailed. Moor claims that, "something miraculous happens when a trail is trailed. The inert line is transformed into a legible sign system, which allows animals to lead one another, as if telepathically, across long distances."

When patterns authors write about what they've found to be used in practice, the patterns they present have the potential to be trails that others eagerly follow. But this potential only exists if the authors explain how to move from one "sign"/ pattern / heuristic to the next. I've seen scant evidence of this. Patterns maps in books typically don't describe movement through the patterns. Instead, like rough hand-sketched maps, they suggest only vague connections. Individual patterns seem more like clumps of potentially interesting waypoints, loosely linked or roughly categorized at best. Most authors stop short of laying out waypoints or "sign" in any specific order to follow. On the other hand, pattern languages, unlike pattern collections, attempt to define one or more sequences of use. Once you add potential sequences, voila! pattern languages seem much more like trails.

I know of few examples of published software design pattern languages. *Object-oriented Reengineering Patterns* by Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz is a notable one. Each chapter starts with a pattern map illustrating potential sequences through the patterns in the chapter based on actions (see Figure 2 for the pattern map for Chapter 4). These maps illustrate small trails with branches, loops, and options. For example, to gain an initial understanding of a design, you can start with either a top down or bottom up approach and proceed until you have enough understanding to move on to your next re-engineering task.

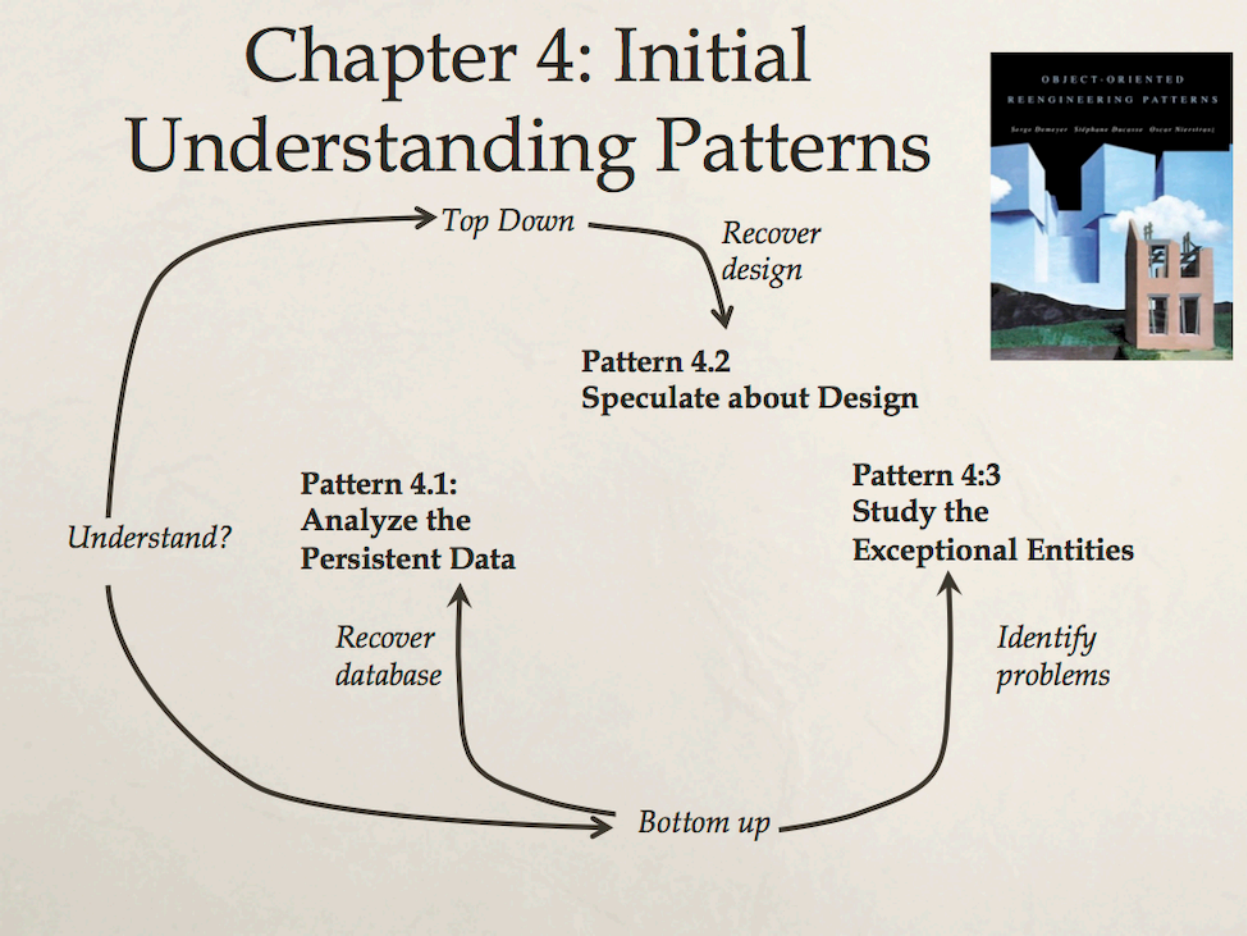


Figure 2. Each chapter in *Object-Oriented Reengineering Patterns* is a small language

Unlike physical trails, where we are guided to move in a singular direction, software pattern languages seem more loopy and fragmented. But unlike a physical trail where we are constrained by the physical terrain, software designers can skip over any pattern they don't find useful or go "off trail" at any point to pick up and apply a useful design heuristic, wherever it is found. It's hard to skip over a part of a physical trail. It's only possible when there's a switchback that you can cut through or a branch. But it is usually those optional stretches away from the main trail and then back again that lead to something really interesting (you don't want to miss that waterfall simply because it is an extra ¼ mile out of the way).

We software designers often invent (design? hack out?) our own tracks. If we don't know what to do next, we become way finders, experimenting and looking around for actions that will propel us forward. To me that doesn't feel like bushwhacking; it just seems expedient. Software designers aren't constrained to follow a patterns trail exactly as any pattern language author suggests anymore than fluent speakers are constrained to express their thoughts using only the formal grammar defined for their language.

So this is where the pattern languages as trails metaphor breaks down. Software design doesn't simply proceed from one known waypoint to the next. It's often more complicated. But sometimes

it is much simpler. We aren't always way finders or followers. Sometimes we are so certain what to do next without consciously following any trail or path or track at all. In that case, the terrain of our software and its design is so familiar to us that we become efficient at just moving through it without much thought. We're not searching for heuristics so much as taking the next (to us, anyway) obvious step.

The roles of trailblazers, travellers, and stewards

> "The soul of a trail—its trail-ness—is not bound up in dirt and rocks; it is immaterial, evanescent, as fluid as air. The essence lies in its function: how it continuously evolves to serve the needs of its users." —Robert Moor

Trails emerge; living useful trails evolve. Wild, ancient trails started as traces—marks, objects, or other indication of the existence or passing of someone or something. Because others followed, some traces over time become tracks—rough ways typically beaten into existence through repeated use rather than consciously constructed. Tracks became trails only when they become followable. And then, with enough following and time and adaptation a trail becomes "alive" with an evolving purpose—it changes and is adapted by its travellers. But this progression isn't inevitable. Traces peter out. Tracks fade from disuse. Trails become lost, abandoned, or fall into disrepair. Still, each at one point in time had utility and served a purpose.

Like trails, through many uses the rough edges of software patterns get smoothed off. If they seem polished enough, and we have enough of them that are related to each other, we who feel compelled to write them down create patterns collections…hoping others find them useful. But unlike physical trails, which change with use and with the weather and the season, our software patterns, collections, and languages aren't so easily changed. Our software design patterns are representations—like maps of a trail; they aren't the trail itself. Consequently, there isn't a direct feedback loop between recorded patterns and how their users have changed them.

If we were careful enough when we wrote down our patterns we also included the context where we found them to be useful. But the context of those who want to follow our trails, is constantly changing with the type of software being designed, the constraints of the larger ecosystem it is part of, and with the skills and tools at hand. Therein lies a big problem for sustaining the liveliness of software patterns. If we want written descriptions to continue to guide others, to evolve and be ever useful, we need to find ways for users of them to refresh them. And that starts by creating vital feedback loops between software pattern users and their various trail keepers or stewards.

> "We tend to glorify trailblazers…but followers play an equally important role in creating a trail. They shave off unnecessary bends and brush away obstructions; improving the trail with each trip." —Robert Moor

We in the software patterns community seem to glorify trailblazing patterns authors. A trailblazer formally identifies a trail by creating marks or "blazes" that others can follow. Most likely, a trail

existed before it was "blazed." But the trailblazer, who made the marks, is credited with creating it. But patterns authors claim to not have created their patterns so much as discovered them in existence and documented them. We are a humble lot. But when pattern authors mark what they see, they make it easier for others to follow. Pattern authors also do a great service in pointing out the features of the terrain, e.g. the design context and forces, as other, inexperienced designers may not consciously think of them otherwise. Indeed, this, too, is a form of trailblazing.

Often confounding to pattern newcomers is the fact that solutions to real-world problems are more complex than the stylized ones written about in any particular software pattern. I have used and extended patterns from several different collections on more than one occasion. I remember feeling when my solutions were more complex and nuanced than the patterns described in these books and that I had found clever ways to extend and augment those patterns. A solution that I worked on that represented complex roles and privileges for individuals belonging to multiple organizational structures far exceeded the simple relationships in the Accountability and Accounting patterns described in Fowler's *Analysis Patterns*.

But I also remember the discomfort of my less pattern savvy colleagues who felt that they hadn't gotten a pattern "correctly" if we needed to refine or extend it. Only after reviewing our design with Martin Fowler and passing along to my colleagues confirmation that indeed, he thought our problem seemed to warrant a more complex solution, did they feel comfortable with our design.

We can get too hung up on the notion that the initial authors of software patterns, e.g. the trailblazers who blazed more visible trail markers and shored up parts of the trail, making it easier for others to follow, are the best curators of their patterns' ongoing evolution. Often they are not. Patterns get modified and refined during their application. It's the pattern users and community of software designers that embrace those heuristics and push them to their limit who discover more useful devices, nuances, modern techniques, and variations.

Unless there is a strong caring community around the original pattern authors, these insights won't get shared with those who care about sustaining those patterns. Even with feedback, renewed versions of "classic" patterns takes sustained energy and attention to detail and the changing software design scene to keep patterns relevant.

Eric Evans speaks of the revitalization of the DDD community which happened when several DDD leaders introduced and explained the relationships between domains, bounded contexts, and the implementation of domain models using CQRS and Event-Sourced architectures.

I spot some hesitancy for some to update "official" trails mapped out by the original patterns authors; not wanting to step on the toes of those original trailblazers. But those of us who want to preserve trails can and should become trail stewards—volunteering to mend and repair and refine those trails we cherish. What we trail followers need to recognize is that not all trailblazers are alike. While certain trailblazers may not welcome updates, others may gladly seek company, advice, and stewardship help. And some trailblazers may have moved on, having passed through their territory and on to newer ventures. Trail followers have just as much collective ownership of the trails they use as those who initially marked them.

Fieldnotes on an experiment collecting heuristics

Motivated to share what I've learned about heuristics and to stimulate others to share and refine their own and other well-known heuristics that might need refreshing/revisiting, I presented a keynote, Cultivating Your Design Heuristics at the Explore DDD 2017 Conference. I hoped to inspire others to take on a more active role as Domain Driven-Design heuristics stewards. The last sentences of my talk abstract had this challenge:

> "To grow as designers, we need to do more than simply design and implement working software. We need to examine and reflect on our work, put our own spin on the advice of experts, and continue to learn better ways of designing."

The day after my talk, I got a Twitter direct message from Mathias Verraes, one of the thought leaders in the Domain-Driven Design Community. My talk had inspired him to get serious about capturing, recording, and organizing his own heuristics. So we met for a couple of hours at the conference.

I was eager to have a conversation with Mathias and share ideas. Mostly I wanted to practice hunting for heuristics through conversation, as well as gain insights into Mathias' personal design heuristics for events. Mathias is expert in event-sourced architectures, an alternative to the "traditional" domain-layering architectures (which includes patterns for storing and retrieving and updating Aggregate Roots into repositories), which Eric Evans had written about in his book (see Figure 3).

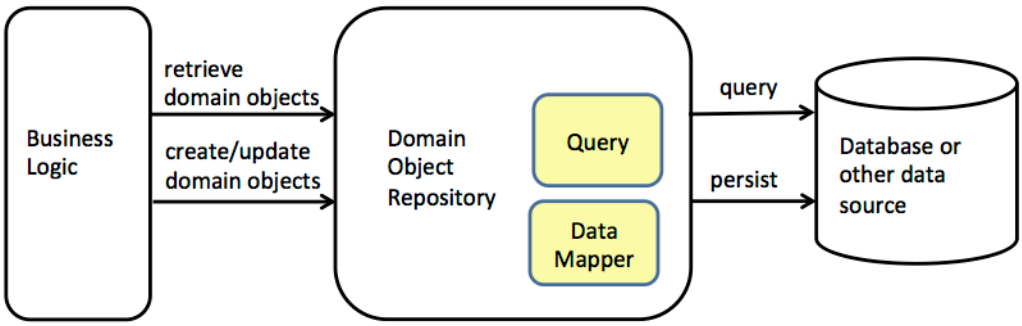


Figure 3. - A layered architecture where business domain objects or aggregates are maintained in a database that is accessed through a repository which hides the data store details from the business layer logic.

In a nutshell, instead of storing and updating Aggregates (e.g. complex business domain objects) into databases, with event-sourced architectures, immutable events are stored with just enough information so they can be "replayed" to reconstitute the current state of any Aggregate. In essence, an event is a record of what the software has determined to have happened. Whenever work is accomplished in the system, one or more "business level events" are recorded that represent the facts known at the time. Events are generated by a software process as a byproduct of determining what just "happened" and interpreted by interested downstream processes, which can in turn, as a result of processing or interpreting the events they are interested in receiving, generate even more

events. Each event is preserved in an event store, along with relevant information about the event. Figure 4 shows a representation of a CQRS (Command-Query-Response-Segregation) architecture, one approach to implement event-sourced architectures. It should be noted that although the figure only shows one event store and one read model, there can be multiple event stores (each representing some cumulative state of the system) and different projections or read models designed for specific queries about those events.

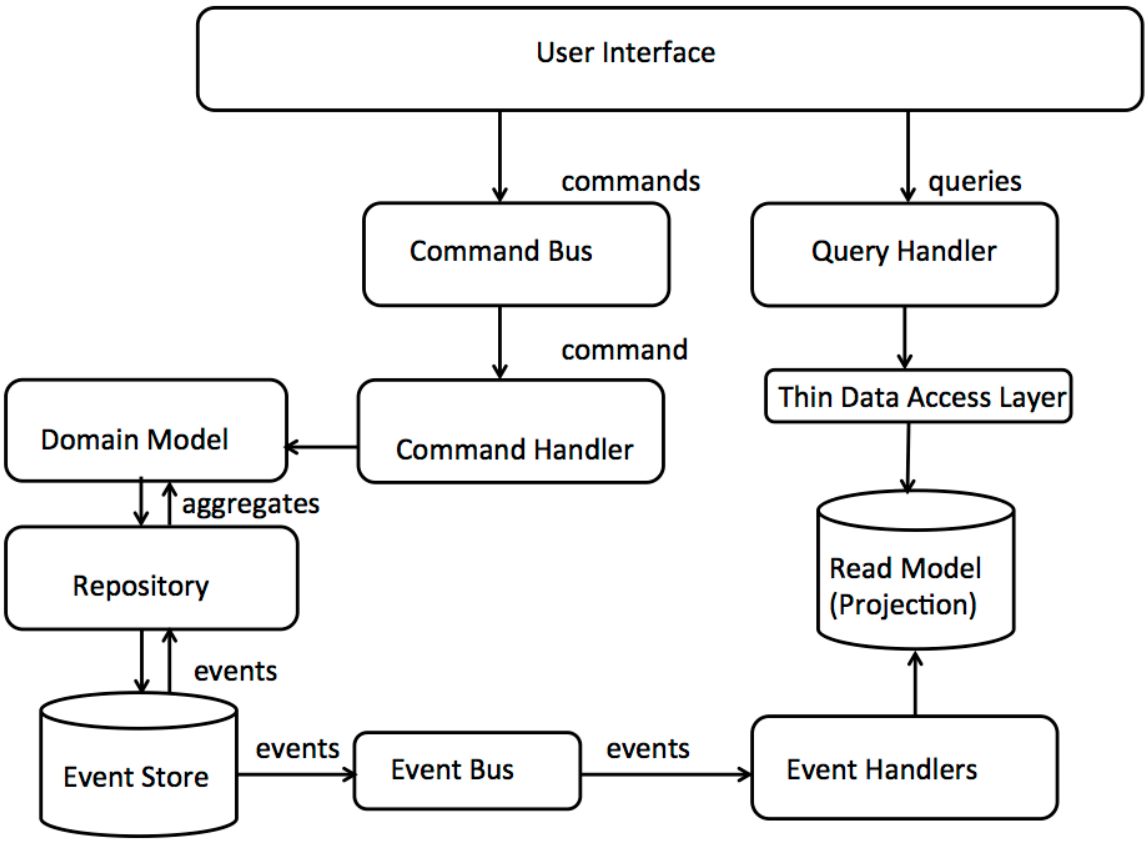


Figure 4. A representative CQRS Architecture

I didn't know Mathias' thinking on designing event-sourced architectures. So I wanted to first ask him to explain some fundamentals before sharing his heuristics for what should be published in an event. Throughout our conversation Mathias used as a working example the designs for car rental, finance, and student grading for courses and modules given by instructors (all examples drawn from real systems he had designed).

Mathias quickly rattled off two heuristics, along with examples:

> *Heuristic:* A Bounded Context should keep its internal details private.
>
> *Heuristic:* Events are records of things that have happened, not things that will happen in the future

For example, an event should be named "a reservation for a car rental has been made" instead of "rent a car" if the customer has just gone online and asked to rent a car. People often confuse what has just happened with real world events that are in the future. When you reserve a car you aren't actually renting it (not yet). You've just reserved it for a future date.

I asked Mathias what he meant by keeping internal details private.

Mathias then shared this example: If you are keeping monetary units in say 10 digits internally in a service, you would only pass out an amount in 2 digits precision because that's all other consumers of the event outside of the Bounded Context would need. Perhaps there was another heuristic exposed by this example.

> *Heuristic:* Don't design message or event contents for specific subscribers to that event.

I wanted to understand the implications of this heuristic. So I asked, "So does that mean that you have to know what processes will consume any event in order to design an event record?" The discussion then got a bit more nuanced. Mathias said that you have to understand how events flow around the system/business. Whatever you do, you publish business events, not technical events that are consumed by other processes outside of a particular Bounded Contexts. So yes, you really need to know how business events might be used to accomplish downstream business processes in other Bounded Contexts. Events along with their relevant information, once published are simply streamed out and stored over time to be picked up (or not) by any process that registers interest in that event. So of course the consumer of an event needs to know how to unpack/interpret the information payload of that event.

Distilling what he said, I offered this heuristic.

> *Heuristic:* When designing an event-sourced architecture understand how events flow around the system/business.

Our conversation continued.

I asked, "Who should have the burden of decoding or translating the event payload into the form needed?"

Mathias answered, "the consumer, of course. But the generator of the event cannot ignore the needs of potential consumers. So there might be an agreed upon standard convention for money, for example, is 2 digits precision."

This led us to conclude we'd uncovered yet another design heuristic.

> *Heuristic:* Design agreed upon standard formats for information in business events based on expected usage.

And just to poke at an edge case that came to mind as we were talking, I asked, "Well, what happens if a new process needs that extra precision?" Mathias was quick to reply, "Well, maybe it needs to be within the Bounded Context of that process that knows of that 10 digits precision."

I pushed back, "But what if it doesn't logically belong in the same Bounded Context?" Which led us to conclude that perhaps there was a competing heuristic that needed to be considered along with the "Design agreed upon standard formats" heuristic.

> *Heuristic:* When designing a payload for an event don't lose information/precision.
>
> That led Mathias to restate that while information within a Bounded Context might contain extra precision or information; information that gets passed "outside" a Bounded Context via a Business Event shouldn't contain "private details." Our conversation continued for over two hours. I have more pages of heuristics notes and examples that I will only briefly summarize.

Me: How much information should be passed along in an event record?

Mathias: Just the key information about that event so you can "replay" the stream of events and recreate the same results. For example, if it is a "payment received event", you don't want to pass along all the information about the invoice that was paid. This led us to some deep discussion about events and time and that time is really important to understand (and that events can be generated by noticing the passage of time, too).

More heuristics tumbled out.

> *Heuristic:* If a different actor performs an action it is a different event.

For example, it is one thing for a customer to report an accident with the vehicle or to return a car, and another thing for an employee to report an accident or even the car itself if it has telemetry to do so. These are all different kinds of events. We discussed more heuristics about events.

> *Heuristic:* If there are different behaviors downstream, then multiple, different events might be generated from the same process.

And this is when Mathias started to draw a representation of his architecture for event streaming and the events that happen over time. He stated that since all events are available to a process, it can find out the "set" of events it is interested in to drive behavior.

> *Heuristic:* Look for a pattern of events within an event stream to drive system behaviors.

For example, you might want to design your system to not send an overdue notice if you've recently received payments, however recent is defined by the business. To do that, the overdue notice process might query the event store for payments to find previous events (and check their timestamps) before sending overdue notices.

We also talked about the situation where a customer changes addresses too frequently (say 3x in a single week). Perhaps this detection of events might cause a fraud detection process to be initiated. And even the same stream of events coming in at a different timescale might represent an opportunity initiate different behaviors/processes.

> *Heuristic:* Consider the timescale when looking for patterns of events. The same set of events over a different time period might be of interest to a different business process.

We reluctantly concluded our conversation when we were invited to join the conference's closing circle. Mathias had written on both sides of a big sheet of paper, sketching ideas as he went. I asked if he wanted to keep the paper. He said no, he knew this by heart as he covers what we had talked about in a three-day workshop he gives. I now wish that I had taken a photo of his scribbling to jog my own memory.

Reflections on the distillation process

This was my very first attempt at actively distilling someone else's design heuristics. I didn't want to bog down our conversation by taking copious notes or interrupting the conversation to stop and record any specific heuristic or tweak the wording of what Mathias said or wrote. So I waited to write up notes about our conversation from memory that evening, inspired by advice I found in Writing Ethnographic Fieldnotes. My goal wasn't to come up with a completely polished pile of publishable heuristics, just a few to get started.

I learned these things from this experiment:

- **Listen**. I need to restrain from sharing my own heuristics and design thoughts in order to let Mathias' heuristics come out. My primary goal was to pick out and follow his trail of heuristics, not mingle them with my own. I'm not used to doing this, so I didn't always silence my internal thoughts enough so I could listen more intently. This will take practice.
- **Let the conversation wander**. It's OK to let the conversation wander to where the person you want to glean knowledge from takes it. But don't let it wander too far away from the topic. It is good to have a design topic around which to focus. Our focus was the design of event records. It wandered a bit to an equally interesting topics, event patterns and time, but since that wasn't our original focus, unfortunately, I didn't capture those heuristics so clearly. The goal is to tease out traces, tracks, and trails of interesting ideas that you want to pursue further.
- **Prepare beforehand**. If you aren't familiar with the jargon around the particular topic, prepare beforehand. I already knew the "classic" DDD patterns and a bit about event-sourced architectures. So I didn't stumble over Bounded Contexts, Event Records, or Aggregates. Someone unfamiliar with those patterns would've had more difficulty following what was said. Trail markers make sense only if you know what you are looking for.
- **Ask questions**. Sometimes I felt like a two year old constantly asking, why, why, why…but I found that uncovering edge cases helped clarify ideas, tease out nuances, and uncover the scope (and certainty) around a particular heuristic. We even uncovered competing heuristics that way.
- **Ask for realistic examples**. Design heuristics grounded in realistic situations are on more solid ground. Use realistic examples instead of made up ones. We don't need to create yet another design for an Automated Teller Machine (ATM).
- **Ask what would happen if?** I did this to gain a better understanding what would happen if the design context changes slightly. Nuances are what makes the process of design so interesting and pattern writing so hard.
- **Go with the flow**. There is no need to stop to record every heuristic in real time. Writing up field notes shortly after our conversation allowed me to be in the moment during our conversation and to ask more probing questions than if I had been pausing to take notes. Perhaps I would've gotten more out of our conversation if I had made an audio recording of it. But I am not certain about that. The "crutch" of having a record might've lulled me into not being so active at remembering and recounting. Sure, I missed some of Mathias' heuristics on modeling time. In hindsight, I think at that point of our conversation I was listening less intently because I thought I knew a lot about time. But also I was getting tired. Active listening is mentally taxing.
- **Photograph scribbles and drawings to jog your memory**. It's easy to do if the person you are conversing with is drawing while they talk. Mathias drew, but he crumpled up the paper after our conversation. So I lost a valuable memento that would have helped me remember his heuristics about time and what constitutes an event. I'll need significant practice if I want to distill heuristics while simultaneously making sketch notes. Oh well. I know I need at least one more conversation with Mathias.

Certainty about the heuristics we distill

Mathias shared several heuristics in a fairly short time. The heuristics Mathias explained were grounded in his direct experience design and building several event-sourced architectures using Domain-Driven Design concepts and patterns. What we discussed was just a taste of what he knows. However, the heuristics Mathias shared were on the whole pretty useful, even though the design of event-sourced systems is a big topic and we jumped right into the middle of it. In hindsight, some heuristics seem self-evident and consequently hard to apply. For example, "don't lose information/precision" seems obvious (if you lost information, then you wouldn't be able to trigger workflows in other components in your system or be able to "replay" events to reconstitute the current state of system things).

The heuristics I like best are those where I can take some specific action and then see whether it results in forward design progress. I don't know exactly what to do with the heuristic, "Don't lose information/precision," other than to verify what each consumer of an event might need. Which leads me to appreciate that event records shouldn't be designed in isolation from their potential consumers. Perhaps I should have restated this heuristic as, "Design event records to convey the precision needed by known consumers of the event."

When I make that wording change then I find that the heuristic, "Don't design information contents of an event record for specific consumers," needs further scrutiny. There's conflicting advice in these two heuristics. On the one hand I can't be overly specific when I design the information in an event record, but if a consumer needs specific information that varies from the typical consumer, what are my options? This seems like a meaty topic warranting further investigation. We briefly touched on this during our conversation, when Mathias suggested, well, if the process needs that extra precision, maybe it needs to be in the same Bounded Context. But I pushed back, saying if it has different behaviors and needs different information, perhaps it belongs in a different Bounded Context.

I remember the heuristic that you might want to generate different events for the same process. But we didn't go into any detailed examples. So how much could I bend that heuristic (is it cheating?) to make it fit this situation?

Other heuristics seem less important—footnotes really. "Agree upon standard formats for information," seems simply good design practice, and not particularly unique to event-sourced architecture. And isn't the heuristic, "A bounded context should keep its internal details private," just another restatement of the more general design practice of encapsulation? Or is there something more significant to Domain-Driven Design's modeling approach? I suspect digging into this topic could lead to another long conversation about Mathias' heuristics for determining what should be in a Bounded Context and under what situations would you refactor, split, or merge Bounded Contexts.

Sure, these heuristics were rough cuts. They need refinement and more details before others who don't have direct access to Mathias will find them useful on their own. Popping up a level, it is apparent that there are a few fundamental concepts that need to be understood before you can understand how to design event records. Unless you know what an aggregate is, heuristics about what information to record from the aggregate root in an event record won't make sense. Heuristics

specific to approaches to designing an aggregate (if there are any) in the context of an event-sourced architecture also need explaining. Which leads me to wonder, what are ways to effectively model an aggregate in event-sourced architectures? Do you make lightweight domain models or something else? How does event-storming, another technique known within the Domain-Driven Design community used to capture flows of business events, fit in?

I know how to model aggregates in a layered architecture, but event-sourced architectures are new territory for me. Are there heuristics for ensuring the business event record contents and various event stores have the "right" representation of information? How should you reference other aggregates or entities within an event record? One answer seems obvious—use a unique identifier for an aggregate… but are there specific heuristics for how these might be generated in an event-sourced architecture? Can I use my prior knowledge of how to do so for more "conventional" architectures?

I am sure we left out some important heuristics, simply because our conversation wandered. A conversation to distill heuristics is not like walking a trail. Interesting waypoints are discovered during the conversation and sometimes conversations wander off into the weeds. What may initially appear important may or may not be nearly so fascinating as it first seems. Conversations, like designs, are not straightforward.

Several have written about event-sourced architectures. Martin Fowler's early blog post on event sourcing[20] laid some conceptual groundwork for event sourcing, discussed what kinds of applications he had found that might be appropriate to use event sourcing, and provided simple code examples. Greg Young's book, Versioning in an Event Sourced System, concentrates on versioning events, a seemingly minor challenge until you get into the nitty gritty design details.

Microservices.io has a single web page on the event-sourcing pattern[21], along with pages for other microservice architecture patterns. Chris Richardson, founder of Microservices.io, has also collected them into the book, *Microservice Patterns: With Examples in Java.* I like the approach taken at Microservices.io where patterns are presented in some detail and readers can ask questions and add comments. Chris Richardson is an active steward of these patterns as well as a trail guide—clarifying points of confusion, directing people to other sources to explore, and trying to get at the real problem that underlies the question that are asked. Some questions led to quite interesting threaded discussions.

Reading the threads on this website, I felt part of a community of fellow pattern travellers on a journey toward deeper understanding. I wanted to hear that answer from other designers who were more experienced with event-sourcing implementations. I liked hearing other designers' voices and learned as much from others' points of confusion as I did from their direct experiences. With design, the devil is always in the details. Conflicting/competing design forces that you need to resolve compel you to make some difficult design decisions.

However, one question (a question I also wanted the answer to) remained unanswered:

> "…I'm trying to figure out how I would apply this pattern to a large CRUD screen where

[20] https://martinfowler.com/eaaDev/EventSourcing.html
[21] http://microservices.io/patterns/data/event-sourcing.html

> the commands mainly consist of Save, Update, and Add for objects with several fields. Thanks!"

Probably, the answer to this is that this it isn't an appropriate situation for using an event-sourced architecture. If you are doing CRUD operations to a database and that database is used by other applications outside of your control or sphere of knowledge, you aren't likely to have a good understanding of how that data is used. So turning an existing, working design on its head to generate events with rich information about the domain doesn't make sense without first understanding how those other applications use and manipulate that information. This may or may not be easy to sort out without doing some serious investigation. On the other hand if your design is simple, efficient and works, why change it to an event-sourced one? There have to be good reasons to make that significant redesign investment.

I'm being transparent about my lack of knowledge to make a point: to keep learning, you have to search for design heuristics that are outside your comfort zone. We become wayfinders when we're in unfamiliar design territory. This learning can be a difficult and frustrating slog when heuristics are scattered, inconsistent, overlapping, or out-of-date. It takes effort to sort out the good bits from the noise, to find and follow any potential tracks. Especially frustrating is when no one else has asked the questions you need answered. When I am not on any trail that others have walked I feel a bit isolated. And yet, I'm not lost. I'm simply on some track. My track. And there's no one ahead of me that I can see. Yet, I know where I've been and I can always fall back to draw upon my more general design heuristics.

More useful information is likely available, waiting for me to bump into it, if only I knew where to look. And if I can't find it, well, I can always experiment.

And yet, how certain can I be about whatever advice I find? I tend to trust patterns authors who put in the time and effort to polish and publish their work, who've spent time marking their trails, checking that others can follow and that their heuristics make sense to other designers. I place high value on the advice of those who've built interesting systems and can tell stories about what they learned including design missteps they made and how they eventually made forward progress. But I don't necessarily throw away what I have found useful just because someone is enthusiastic about a new software design approach and a new-to-me set of heuristics. They may be experts at some software design approaches that takes years, if not a lifetime, to master. At best, I might only be able to clumsily apply their heuristics after some concerted effort. Or alternatively, they may be trailblazers to places where I don't want to go.

Techniques for actively cultivating design heuristics

We each have our own set of heuristics we've acquired through reading, practice, and experience. Our heuristics, like living trails, continue to evolve and get honed through experience. Some of our heuristics prove durable and are still useful, even in new design contexts. For example, for me, using the lens of role stereotypes from Responsibility-Driven Design to understand system

behaviors is useful, even though I know there are newer stereotypes for functional designs and Internet applications that people are writing about. Characterizing roles and interaction patterns is a useful heuristic for understanding the designs of many systems. I never thought the original stereotypes I conceived to help me and others understand object-oriented designs were universal (and all the stereotypes there were). So I welcome new ways of characterizing design behaviors.

Some heuristics we discard because our design constraints radically change. I no longer worry about managing memory footprint and have put aside those heuristics that were useful back when I designed systems that required memory overlays—for me that trail has been long abandoned. Other heuristics get pushed to the back of our minds when we find new or trendier heuristics we like better. When I discovered object-oriented techniques, I put aside other approaches to structuring systems because I found objects to be so useful. Long ago I took a decision to head down that trail and have continued on that journey.

To keep learning, we need to integrate new heuristics with those we already know. Billy Vaughn Koen cautions us to not judge our earlier designs (or earlier designers) too harshly against today's design standards. Collectively, our state-of-the-art (or SOTA) keeps progressing. And as active, engaged designers, so do we. Recently I have been exploring functional programming languages and designs that employ them, simply because I want to compare heuristics for designing these systems with older, more familiar-to-me ones. I don't want to get stuck in a rut. Although I may not become an expert, I'll be a better designer with a richer set of tools.

Recording "Sign" with Question-Heuristic-Example Cards

I have also experimented with ways to articulate new-to-me heuristics in order to see how they fit into my heuristic gestalt. I've been playing around with using index cards as a means to capture the gist of a heuristic. This simple technique structures a heuristic in three parts: a question, the answer (which can be then polished into a formulation of the heuristic), and an example or two to help me remember. I call them QHE or "Q-Hee" cards, for the lack of a better name (see Figure 5). This use of index cards to capture design heuristics is inspired by CRC (Class-Responsibility-Collaborators) design cards invented by Ward Cunningham and Kent Beck.

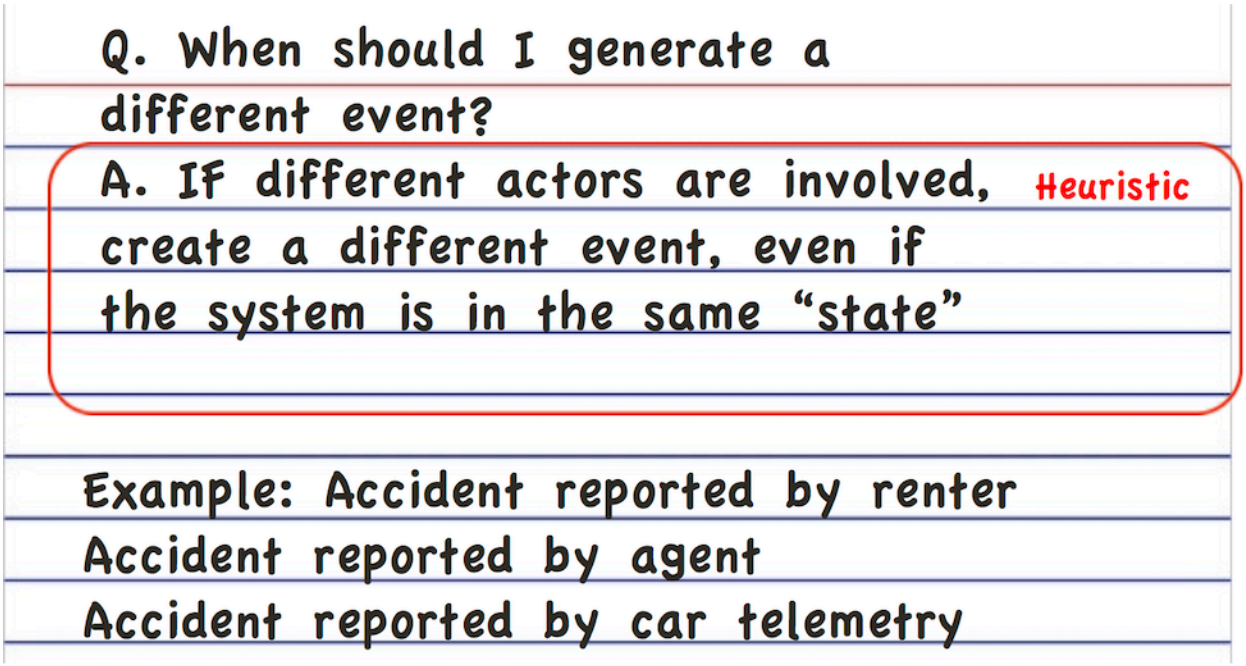


Figure 5. Following this heuristic, 3 different events would be generated because there are 3 different actors.

An advantage of QHE cards is that they are easy to write.

But just like CRC cards, they can be too terse.

Without actively integrating the heuristic captured on QHE card into my design heuristic gestalt, I find it quickly loses meaning. Once I convert these to a richer form (either by writing further about the heuristic or sketching out a more detailed design example or writing some code), I can then recall more subtleties about that heuristic.

Distilling what you hear

One way I can more actively learn is to view technical presentations as opportunities distill what I hear and integrate those heuristics with my own. I discovered that if I take a picture of some interesting speaker and/or a slide they were presenting it served to jog my memory. Looking at the picture helps me remember what they said so I can write up field notes, if I choose, long after the presentation.

Here are two photos I took at the DDD Europe 2018 conference.

The first is of Eric Evans telling us the story of how he goes about exploring a design concept and all its limitations and design surprises. I found each line on the slide to be a personal heuristic Eric uses to do this (the rest of his talk was filled with examples exploring the quirks and complexities of date and time).

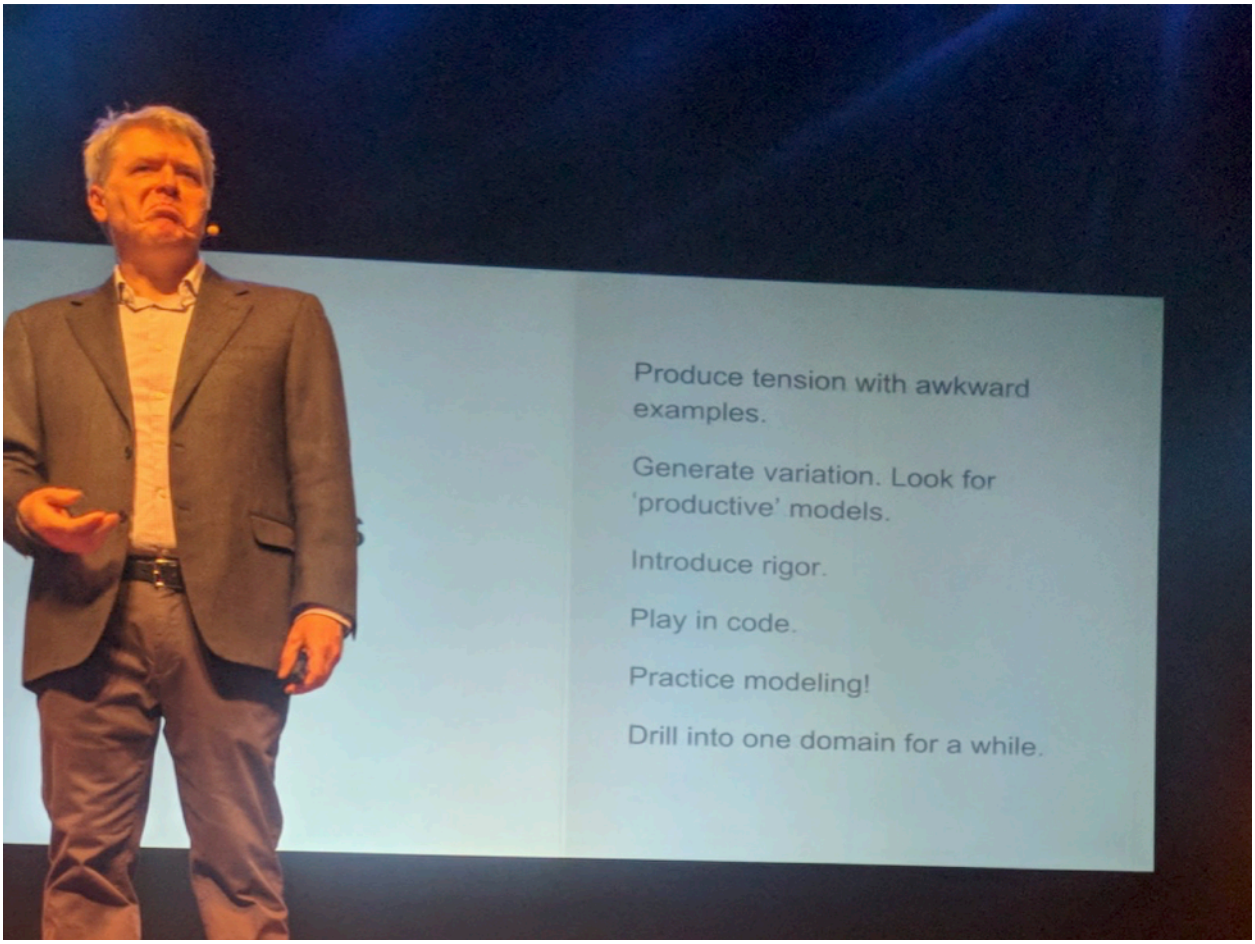


Figure 6. Photo from Eric Evans' keynote at Domain-Driven Design Europe 2018 introducing how he understands a domain

The next photo is from a talk by Michiel Overeem on versioning event stores, a fundamental element of event-sourced architectures (see Figure 7). This slide summarizes the various approaches Michiel found when he surveyed other designers. Event stores are supposed to be immutable. You use them to play back events and recreate system state. Conceptually they are write once stores. But if your event schema changes, various components need to then be able to interpret these new event structures. But if your event schema changes, various components need to then be able to interpret these new event structures. So how do you make that work in practice? You select a versioning approach depending on a number of factors including the size of your event store, your ability to process extra information on event or to transform on the fly to a new format, and your event store update policy.

While Michiel eventually put his slides[22] online, this photo was enough to jog my memory and make the connections between heuristics for updating event stores and heuristics I'd written in pattern form for updating Adaptive Object Model (AOM) systems. Although Event-sourced and Adaptive Object-Model systems are quite different architecture styles, they have similar challenges

[22] https://speakerdeck.com/overeemm/dddeurope-2018-event-sourcing-after-launch

with updating their models' schemas.

Figure 7. Photo of summary slide from Michiel Overeem's presentation on Event Sourcing After Launch

Sharing Heuristics to Start Conversations

Since my initial conversation with Mathias, we've both become energized to do more heuristics hunting. This led to a one-day Heuristics Distillation workshop I held at DDD Europe 2018. At that workshop I shared my heuristics journey and then participants shared a few of their cherished heuristics. Since then, I've given other presentations about design heuristics and have been encouraging others to articulate and share their heuristics. Consequently, Victor Bonacci held a workshop at Agile 2018 on Coaching Heuristics: What's in Your Toolkit? Coaching heuristics aren't software heuristics, but in Vaughn Koen's definition of design heuristics, they do fit: Any thing we do in an attempt to make forward progress towards a goal.

The format of Victor's workshop was quite effective. First he explained what heuristics were, then showed a slide listing the 54 coaching heuristics he had collected over the years organized by category. Victor has also created a card deck for his heuristics. He doesn't sell these cards, but

gives them out as gifts. On the face of each card is an illustration or phrase and on the backside the name/source of the heuristic. He finds the deck a useful way to jog his memory as well as means of sharing just the gist of their idea with others (see Figure 8).

Teaching each heuristic in this long, long list would have overwhelmed us. Instead, Victor quickly introduced two or three heuristics in a particular category and then gave us a situation to briefly discuss in small groups. We also had a deck of Victor's coaching heuristics to refer to if we wanted. We discussed what heuristics (our own or others we had heard about) we might use to try to improve the situation. After each round of discussion, a few shared what they had talked about with the larger group. We repeated this cycle three or four times, learning a few more of Victor's heuristics, but also, more important it seems, sharing our experiences and our own heuristics. Although the format of this workshop was similar to that of patterns mining workshops, it wasn't focused on capturing these heuristics so much as it was getting people to share their experiences with others.

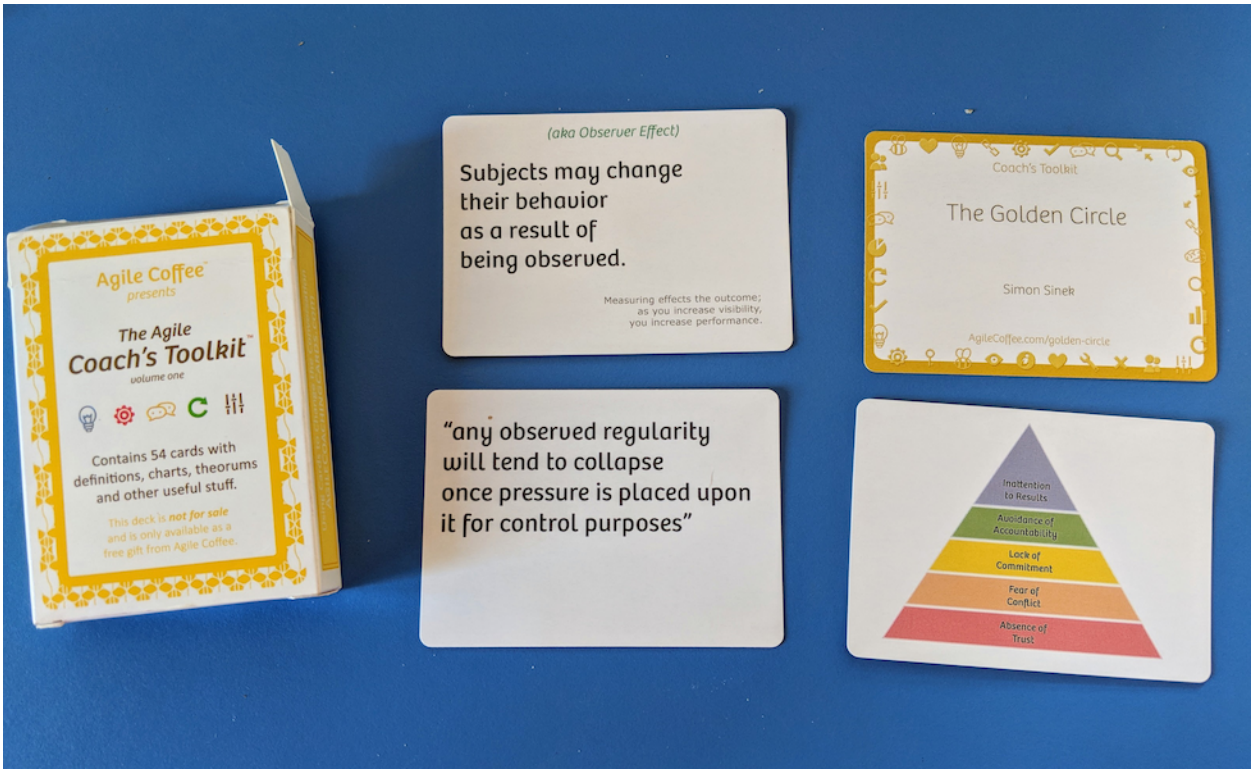


Figure 8. Coaching Heuristics Cards created by Victor Bonacci. Each card carries the gist of the heuristic, either as a drawing or phrase on the front side, and the name and source on the back.

Holding an Imaginary Debate

One way to appreciate another designer's approach is to walk a mile in their shoes. Barring that rare opportunity, an intriguing alternative is to take some design advice you find and imagine having a thoughtful debate with that designer. Counter their advice with an opposing set of arguments. Then, distill the essence of the heuristics you find in both your arguments and reflect on the relationships between the heuristics embedded in each point of view. I find this is easier to do if you have either

a strong negative or positive reaction to some particular bit of advice. Surprisingly, arguing for an approach that differs from your preferred design heuristic helps you gain an appreciation for that perspective.

For example, Paul Graham, in an essay Revenge of the Nerds[23] writes,

> "As a rule, the more demanding the application, the more leverage you get from using a powerful language. But plenty of projects are not demanding at all. Most programming probably consists of writing little glue programs, and for little glue programs you can use any language that you're already familiar with and that has good libraries for whatever you need to do."

One counterargument to Paul's thesis might be, "What you recommend for complex systems makes sense—use a powerful programming language. But if I am not in a time crunch and what I'm building is simple, I shouldn't always take the easy path. If always I took your advice for simple programs, how would I ever learn anything new? If the problem is simple, that might be the perfect opportunity for me to try out new ways to solve it and learn something new especially when the consequence of failure isn't high. Also, sometimes what appears to be simple turns out to be more complicated. And when I push on the limits of what tools and frameworks were designed to do, it is important to stop and rethink my current approach instead of trying to hack away at it until I patch together a solution. Or at least take a break before coming back to what I've been struggling with."

Two heuristics distilled from Paul Graham's advice:

> *Heuristic:* Use powerful programming language/toolset hand when you have a demanding design problem.
>
> *Heuristic:* It doesn't matter what programming language you use if you have a simple program. Use programming languages, tools, and frameworks and libraries you are familiar with.

And the three heuristics found in my counterargument:

> *Heuristic:* Use simple design tasks as an opportunity to learn new design approaches, tools, programming languages, and frameworks, especially when you aren't in a time crunch.
>
> *Heuristic:* When you find yourself constantly fighting against the common usage of a framework, revisit your current design approach.
>
> *Heuristic:* Take a break when you have been working too long and don't feel like you are making progress.

[23] https://www.eecis.udel.edu/~decker/courses/280f07/paper/Revenge.pdf

On reflection, Paul Graham's advice seems geared towards designers who find they waste too much time trying new tools and techniques instead of implementing workable, familiar solutions. On the other hand, without stretching and trying something new, designers can get stuck in a rut. Both viewpoints have some validity. There are always competing heuristics to choose from. And depending on your current context, past experiences, and preferences, you decide between them.

The work of reconciling new heuristics with your SOTA

Sometimes it takes effort to first understand and then reconcile newfound heuristics with your existing ones. Designers use different terms to describe similar (but not identical) concepts. Mapping others' terminology to your language can be fraught with uncertainty.

To illustrate this difficulty, I took advice from Daniel Whittaker's blog post on validating commands in a CQRS architecture[24] and tried to align his heuristics with mine for validating input from an http request.

My heuristics for validating input are roughly as follows:

> *Heuristic:* Perform simple edits (syntactic) in browser code.
>
> *Heuristic:* On the server side, don't universally trust browser-validated edits. Reapply validation checks when receiving requests from any untrusted source.
>
> *Heuristic:* Use framework-specific validation classes only to perform simple syntactic checks such as correct data type, range of values, etc.
>
> *Heuristic:* Use domain layer validation and constraint enforcement patterns to validate all other semantic constraints and cross-attribute validations.
>
> *Heuristic:* Value consistency over cleverness when performing validations.

I also make the further distinction between descriptive, operational state, and life-cycle state attributes, based on concepts found in *Streamlined Object Modeling*. Some domain entities go through a one-way lifecycle, from initial to a final state. The current values of any of their life-cycle attributes determine permissible state transitions. In a traditional architecture, the current state of a domain entity is retrieved from a database via an appropriate query. In an event-sourced architecture the current state of a domain entity is synthesized by replaying all of its events (if this is expensive to do, the state may be cached). Some entities switch between different states, which are represented either directly in a state attribute or synthesized through determining current values of its operational attributes. The state such an entity is in determines how it behaves.

In his blog, Daniel uses different words to describe different kinds of data validations. He speaks of "superficial" and "domain" validations. Are these the same as my "simple, syntactic" and "semantic

[24] http://danielwhittaker.me/2016/04/20/how-to-validate-commands-in-a-cqrs-application/

constraints"? Daniel characterizes "superficial" validations as those constraints on input values that must hold true, regardless of the state of the domain and gives this heuristic:

> *Heuristic:* Perform superficial validations before issuing a command, ideally on the client side as well as the server side.
>
> He also characterizes some validations as being "superficial but requiring the lookup of other information" and advises:

> *Heuristic:* Perform "superficial validations requiring lookup" in the service before issuing a command.

Finally, he speaks of "domain validations" where the validity of a command is dependent on the state of the model (or I might restate, the current state of the domain) and recommends they be validated in the domain object:

> *Heuristic:* Perform domain validations in the domain objects.

It seems clear that I must do some mapping of his concepts to mine in order to make sense of both sets of heuristics. Alternatively, I could let these different sets of heuristics rattle around in my brain without making any attempt to integrate them. But that might lead to "parroting" the new heuristics without really understanding how and where to apply them or worse yet, ignoring.

When is it worth the effort translate heuristics from one language of design thought to another and then reconcile them? I suspect that this question isn't asked often. When faced with a new design challenge and new techniques, we have to absorb them the best we can or we won't be able to jump into that new way of designing. When a design approach is so radically different from what we know, it's easier to absorb new and different terminology.

It's when concepts overlap that it takes more effort.

There seems to be an overlap between what I describe as syntactic validations and what Daniel calls superficial validations. But "superficial but requiring lookup of other information" doesn't directly map to any concept I know of. I can conjecture what it might entail. "Superficial but requiring lookup of other information" could roughly correspond to my cross-attribute constraints (where the knowledge of what to look up seems to be located closer to domain logic, as in a domain service). And his "domain validations" seem to overlap with operational and life cycle state attributes as well as other cross-domain attribute checks that don't require any "lookup".

This mapping isn't perfect. But it will suffice. My heuristics are at a slightly different level than Daniel's. For example, I speak of how to use frameworks for simple validations. I also include a heuristic for generally how to approach validation design (value consistency over cleverness). But after performing this mental exercise, I think that I understand his heuristics well enough to integrate them with my own.

In retrospect, this wasn't that hard.

But still, it took some effort.

Dealing with uncertainty, conflicting heuristics, and details

I suspect we need to let go of some design certainty before we can truly learn from others. When we are so certain we run the danger of painting ourselves into a corner when there are better paths we might take if only we hadn't been so certain of what we were attempting. Yet it's not always appropriate to be experimenting. Sometimes we are better off if we keep to a well-trodden design trail. But first we need to know how to find that trail.

Most of us, most of the time don't start designing every day from scratch. There are usually many constraints already in place. Our task is mostly that of refining some design aspect of a pre-existing implementation. In that case, we jump in and get to work, with more or less certainty based on where we've been, what we know about the existing design, and what the task is ahead of us. We may not even know what trail we are on, just where we are at the moment.

Even so, we still need to make decisions. And those decisions can have far reaching impact. And as we do, we should be aware that multiple design heuristics are always in competition with each other. Should we leave that working code alone or refactor it (not knowing where it will lead – but hopefully to a clearer design)? Should we apply the heuristic, "only refactor when you are adding a new feature" or stop when we notice the code growing crufty and poke at its design (XP calls this it a design spike)? If we don't, we may be working at refining a shaky design that eventually drags us down. This is how technical debt grows.

You can always find a bit of folk wisdom to support what you want to do; and another equally pithy one advising you to do the exact opposite. For example, see Proverbs that Contradict Each Other[25]. Our challenge as designers is to sort through competing heuristics and make a coherent design.

Michael Keeling and Joe Runde recently reported on their experiences instilling the practice of recording architecture decisions[26] into their team. Initially, Michael hoped that simply by recording decisions, this would lead to more clarity about their existing designs and improve overall systems' quality. When designers record their decisions, they lay down a track for others to follow, and to retrospectively learn from. Each decision is "sign" along a unique design journey. Although initially

[25] https://www.psychologytoday.com/us/blog/the-human-beast/201202/proverbs-contradict-each-other
[26] https://www.agilealliance.org/resources/experience-reports/distribute-design-authority-with-architecture-decision-records/

it might be hard to sort out what decisions are worthy to record and to get people to actually write them, eventually there is a payoff.

If I accept that software design is always filled with some degree of uncertainty, any mark or track I lay down to show where I've been (even better if I include what I was thinking when I made a design choice) helps me and others around me support our design's evolution. These decisions could lead to collectively shared awareness of design choices and become the basis for creating a well-followed trail of collectively shared design heuristics. At the very least, those decisions over time create "sign" that others can trace backwards to better understand why the design currently is the way it is.

And I suspect that written design decisions might lead to more commonly shared heuristics, even if they aren't recorded.

As patterns authors, we intentionally create waypoints—our patterns are points of interests along a design trail we hope others can traverse. But we shouldn't be content to only write in pattern forms. Patterns convey critical information so that others on similar journeys can learn about our design thinking. But I think we have an opportunity to offer our fellow designers much more.

What if we were to tell more of our personal story as designers and pattern makers? We might describe what territory we've passed through, what systems we've designed or seen, and under what conditions they were designed. Or, we might share how we discovered our software patterns and enumerate other potential waypoints that spotted or were aware of but didn't include (and why). We might share where we'd like to travel—other design contexts where we are curious, or not—places where we are cautious or reluctant to recommend using our patterns. We could experiment with recording other heuristics that fill in the gaps, conflict with, augment, and mesh with our patterns. We might share how confidant we were about our patterns' utility or our perception of their relative value and whether that has changed over time. Or we might be so bold as to rate our pattern trails with recommended design experience required to traverse it successfully. While all this stuff is "outside" our patterns, I think it this is important information for designers to know.

And yet, patterns are just a small part of a much larger body of design know how. Heuristics, like patterns, can be expressed at various levels. Some are small, simple acts. Others are bigger steps, taken at the beginning of a design journey. There are so many design heuristics. We pattern authors can't hope to mine, organize, or write about them all. Nor should that be our goal.

Each designer has a wealth of heuristics she has internalized yet may have difficulty explaining to others.

But something magical happens when you formulate a heuristic in your own words and share it with another. It is in the telling to another that I clarify my thoughts. And when I am able to patiently answer their questions, I find I gain even deeper insight.

If I take time to write down a heuristic, the act of creating a personal memento brings me even more clarity. And when I've revised my writing, shared it and gotten feedback as to whether they understood and appreciated my heuristic (at least a little) then I have something I can share with others separated from me by time or space or distance.

This progression from doing to explaining to recording to effectively communicating can be difficult.

Not all heuristics are significant enough to warrant a lot of time or energy polishing them. But those that seem important to you are worth sharing. And in conversation, you just might find that what you thought was a simple and obvious seems profound to someone new to your well-trodden design trail. And if you're lucky, they might even share an insight or observation that adjusts your thinking. As long as we keep learning from each other, design will continue to be fun, and equally important, we designers will continue to evolve our state-of-the-art.

Ubiquitous Language - More Than Just Shared Vocabulary — Avraham Poupko

A key tenant of DDD is *ubiquitous language*, and rightfully so. DDD is about improving domain communication and understanding. What better way to improve understanding than to make sure that we all speak the same language, and use the same words to describe the same things? In this paper, I use some insights and concepts from the study of how language is used, to explain in some detail how it is that ubiquitous language provides shared understanding.

The term ubiquitous language is often used to mean "shared vocabulary". It is important to put emphasis on the word *language* in "ubiquitous language" to make the point that we need to expand our understanding of the term ubiquitous language to include not only the words we use, but how we use them.

The importance of language

Language is so much a part of our lives that we sometimes tend to overlook its power and complexity. We just assume that language is there so that we can talk to each other. The truth is that the use of language is one of the most complex human activities. And proper understanding of language allows us to gain deep insights into our humanity. Here is a quick overview of why language is so important to us.

Cooperation comes from communication

Any non-trivial man-made system is always the result of cooperation. No system of any complexity has ever been created by one person. Cooperation cannot happen without communication and for communication to happen we need language. We need to have a way to communicate thoughts so that when I have a certain idea or thought in my mind, I can invoke that thought or idea in your mind. Once we have the same thought, the same vision of what is, what can be and how we get there, we can start cooperating.

Theory of Language

To better appreciate the importance of ubiquitous language, it is worth while diving a bit more into some language theory and how language is used. Particularly, I would like to focus on how we use language to communicate complicated and complex ideas.

Words vs. Concepts

Let's call those ideas that we wish to communicate to each other *concepts*. Words themselves are not concepts, but our brain is able to translate words into concepts. Even though people write and talk in *words*, they think in *concepts*. We know this to be true because people who cannot use verbal or oral language (such as infants, people that have suffered a traumatic injury or blind-deaf people that have not been exposed to language) are still capable of thought. They may struggle to communicate and express these thoughts because they do not have language, but they certainly have these thoughts. When I struggle to find words to properly explain an idea, or when I say "words cannot express what I feel", I am saying that there are some thoughts or emotions that are very real, but that do not easily lend themselves to verbal expression. It might be true that language is critical for developing extensive concepts and abstract thinking, nonetheless many cognitive scientists agree that we do not need language in order to think.

Because we do not know how to communicate in concepts, we need a "communication layer". I think something, imagine something or even feel something which I communicate to you in words. I hope that when you hear those words, the image I have in my mind will be invoked in your mind. In order for this to happen effectively, we need to share a common mapping of words to concepts. The more consistent we are with the way we use words, the more likely it is that the mapping from words to concepts is as expected.

Language is more than just words

On their own, words are sequenced in a rather one dimensional way. When speaking or writing, a word can come before or after other words. How is it that we can communicate such rich and complex ideas using only words? More so, words are relatively simple to utter and to understand. Thoughts are complex. How then do we communicate complex thoughts using simple words? The main answer is *syntax*. It is syntax that allows us to combine simple words together and to express complex thoughts. In the words of the great language scholar Steven Pinker: *Syntax is complex, but the complexity is there for a reason. For our thoughts are surely even more complex, and we are limited by a mouth that can pronounce a single word at a time.*

Simply put "syntax" is "the set of rules, principles, and processes that govern the structure of sentences". However, that is a gross oversimplification. It is syntax that allows me to express arbitrarily complex thoughts and ideas using simple words, just by organizing them. Consider, for example the simple yet profound statement by Kafka: "A wet man does not fear the rain." This is a statement about human nature that uses imagery, symbolism and metaphor yet is comprised of very simple and well known words. So, when we use a ubiquitous language, we are not only agreeing on the meaning of words, but we are using syntax to communicate the complexities of the system that we are creating.

Syntax provides for the exchange of complex ideas

We can imagine a group of creative people that want to exchange ideas with each other, and perhaps come up with new ideas. As the conversation goes on, the ideas that these people are discussing

are becoming more and more complex. If they all speak the same language and share the same vocabulary, then they can express extremely complex ideas using syntax. However, if they do not speak the same language, or do not share the same vocabulary, these people are extremely limited in what they can express, and syntax won't help. While a shared vocabulary is a critical element needed for the discussion of complex ideas, it is syntax that actually enables the complex communication to take place.

Let's say for example that in our domain we use the concept of "invoice". This is a real world concept that as a noun means a demand for payment for a service rendered or goods delivered, and as a verb in means the delivering of the demand to a customer. But there is quite a bit of nuance here. Even as a noun, "invoice" can mean the demand for payment, or it can mean the paper or email representing the demand. I can say "we need to create an invoice for that customer" and mean many things. It can mean that we need to print up the paper, or that we need to calculate how much he owes so that we can demand the payment. When designing billing software, "invoice" might be a database record representing the sum owed, or a PDF that we need to mail to the customer, or it can mean the actual email that we mailed. We will need to find words for an invoice that has been created, but not sent, and an invoice that has been sent, but not settled, and an invoice that has been settled. The more we talk about invoices the more we will understand each other. So when I say: "I think we will not be ready to deliver 'invoice'. The creation of the invoice fails if one of the line items contains a reference to a different currency", the rest of the team members will know exactly what I mean.

Learning language is about learning syntax

Watching children learning to communicate we observe something astounding. Children learn new words from their environment, and are then able to use that vocabulary to articulate complex ideas using sentences that they have never heard before. This continues throughout our lives. We use a fixed and limited vocabulary and a fixed set of syntax rules to utter sentences that are truly unique. Each and every one of has said countless sentences that have never before been said, and might never be said again. Here too we see that while we need common words to express our creativity, our expression of creative thought happens mainly through syntax. As designers and developers, when we are able to express *new* ideas and create *new* sentences using the agreed on ubiquitous language, we can say that we are masters of the language, or at least proficient in it.

Levels of Communication

Words have multiple meanings. I am not referring to homonyms and homographs (words that sound alike or that are spelled alike but have different meanings). I am referring to the same word with its dictionary definition, but that invokes a different image, idea or concept. We might use the word *semantics* to refer to the meaning of the word.

Words have at least three levels of meaning.

Denotation

Denotation is the explicit literal meaning as defined by a dictionary. (Sometimes called “a direct representation of a simple real world idea”).

Connotation

Cultural and emotional meanings that a word might have. These are sometimes taken to be positive or negative.

Association

Words will also evoke certain thoughts or ideas by association. These are not necessarily cultural, rather they are dependent on the particular experiences of the speaker and audience. When we use language to discuss the complexities of the system, we are using all levels of meaning. So we are not just using words with their dictionary denotation, we are also using connotation and association. It is important that when we use the same words, we are aware of those different levels of meaning, because it is those levels of meaning that allow deep and complex conversation to take place.

“Invoice” as an example

For example, lets look once again at the noun “invoice”. Its denotation is a “demand for payment” or perhaps a digital representation of a demand for payment. But that might not capture all the nuance of the word invoice. Reading that definition, I might feel that I only issue an invoice of the customer does not pay on his own. The connotation of the word “invoice” might be the knowledge that customers do not pay until invoiced. Not because they are trying to get away without paying, but because that is how business works. An invoice is not something you issue after the customer has refused to pay, it is something that every transaction must have in order for the business to run properly. In my particular group the word “invoice” carries an association of criticality and complexity. If the invoicing has a mistake, the business suffers. If the invoice does not link properly to the audit trail, we might all be called in for investigation. This association was created in part when I delivered an invoice module with a small flaw that miscalculated the tax due. I lost customer trust, missed a chance at promotion, and was lucky I did not get fired. So when my team hears the word “invoice”, we get serious.

As you can see, when we talk about invoice, there are many levels of meaning at work that come together and allow us to create a common understanding of what “invoice” means.

A dictionary is not enough

How do we use language?

In the world of system design, language is not only used to *describe* the system, it is used to *discuss* the system. This is an important distinction. When I am describing a system, I have a clear image

in my mind, and I am using language to describe it to you, so that you will have that image in your mind. However, when we are discussing a system, we each have a different perspective or understanding of the system as well as a different mental model. As the discussion progresses, the two models start converging to one model, and the meaning and uses of the words at all levels start converging as well. The more we discuss the model using the same words, the more those words will evoke the same images in our minds.

Learning a new language

How do we learn a new language? A real bad way to learn a new language is to learn from a dictionary. The dictionary will teach you the meanings of words, but it will not teach you how to use those words properly. In order to really learn a language, you must talk with other people who use the language and see how they use the words in different sentences and different situations. You must try speaking words yourself and see how other speakers of the language respond to the way you use them in sentences. You will see what syntaxes are useful in conveying your meaning. If you are part of a group or community, you will discover that over time, words start developing their own context-dependent connotations and associations.

The same applies when we are learning a new domain. Learning a new domain is not only learning the domain objects and the relationships between them. As Eric Evans points out, learning a new domain includes learning the proper names for the domain objects and relationships. I argue that it includes even more. Learning a new domain also means the evolving of a language in the wider sense of the word. A language that includes all the connotations and associations, as well as syntax rules that allow us to have deep and meaningful conversations about the domain.

In summary

Domain-Driven Design has truly revolutionized the way we think and talk about domains, domain problems and the solutions to those problem. A critical component in Domain-Driven Design is ubiquitous language. The power of ubiquitous language is not just in giving a common meaning to words, it is in being aware of and making use of the complexities of language. If we understand those complexities, we will be better communicators. I believe that a deeper study of the theory of language by Domain-Driven Design practitioners will advance Domain-Driven Design. By better understanding how human communication works, we can better communicate. Better communication always leads to better design.

Domain Modeling with Algebraic Data Types — Scott Wlaschin

Authors note: It is a privilege to be able to contribute to this project. Even though it has been fifteen years since 'Domain-Driven Design' was published, its insights and wisdom are just as valuable as ever, and continue to influence each new generation of programmers. One thing that has changed since it was written is the rise of functional programming as an alternative paradigm to object-oriented programming. Many articles on functional programming focus on the mathematical aspects, but I believe it has great potential for effective design in conjunction DDD principles. This chapter, condensed from my book "Domain Modeling Made Functional," explains why.

One of the main aims of Domain-Driven Design is to promote good communication between the development team, domain experts, and other stakeholders. In particular, there should be a very close correspondence between the mental model of the domain experts and the code used in the implementation. That is, if the domain expert calls something an "Order" then we should have something called an "Order" in the code that behaves the same way. And conversely, we should avoid having things in our code that do *not* represent something in the domain expert's model. That means no terms like "OrderFactory", "OrderManager", "OrderHelper", etc. Of course, some technical terms will have to occur in the codebase, but we should avoid exposing them as part of the design.

So the challenge is: how well can we create code that matches the domain? Can we create code that reads like text and is understandable by non-developers? And can we avoid introducing non-domain terms like "Manager" and "Factory". I think we can do all of these things by using *algebraic data types* for domain modeling.

Discovering common concepts in domain modeling

When we talk to domain experts during a discovery phase, we will encounter verbal descriptions of the domain such as these:

- "An order line has an order id, a product id, and an order quantity."
- "Contact information consists of a personal name and a contact method, where a contact method is either an email address or a phone number."
- "An email is either validated or unvalidated. Password resets should only be sent to validated emails."
- "A purchaser is either a one-time 'guest' or has been registered on the website. A registered purchaser has been given a customer id, which a guest doesn't have."

- "To pay for an invoice, you start with an unpaid invoice and payment information, and you end up with a paid invoice."

As we build our Ubiquitous Language from these discussions, there are some common patterns that occur, which we might classify like this:

- Primitive values
- Groups of things treated as one
- Choices between things
- Workflows (a.k.a. use-cases, scenarios, etc)
- States and lifecycles

Let's look at each of these in more detail.

Primitive values

Domain experts will never talk about "integers" or "strings". Instead, they will use domain concepts such as "order quantity" or "email address". These concepts may be *represented* by an integer or string, but they are not equivalent. For example, an "order id", a "product id", and an "order quantity" may all be represented by integers, but the domain concepts are in no way equivalent to an integer. It doesn't make sense to multiply an "order id" by two, for example.

And even concepts which *are* integers don't correspond directly to an `int` in a programming language — there are almost always constraints. For example, an "order quantity" must be at least one, and probably has an upper bound as well, such as 100. It's unlikely that a sales system would let you order 2 billion items!

Similarly an "email address" and a "phone number" might both be represented by strings, but again they are not interchangeable, and each will have special constraints.

If we were to document some of these primitive values, we might make notes like this:

```
data OrderId              // opaque -- don't care about representation
data ProductId            // opaque -- don't care about representation
data OrderQuantity is int // constrained to be between 1 and 100

data PersonalName is string // Must not be null or empty. Must be less than 100 char\
acters.
data EmailAddress is string // Must not be null or empty. Must contain @ symbol.
data PhoneNumber is string  // Must not be null or empty. Must only contain digits, \
parens or hyphens.
```

When we come to translate these notes into code, we will want to preserve the simplicity of these descriptions. The code should represent the domain as closely as possible.

Groups of things treated as one thing

Of course, some domain values are not primitive, but are composed of other smaller values. To document these kinds of things, we might make notes using "AND" to group the values together, like this:

```
data OrderLine is OrderId AND ProductId AND OrderQuantity
data ContactInformation is PersonalName AND ContactMethod
```

Choices between things

Another common pattern is that a concept is comprised of alternatives.

- "An email is either validated OR unvalidated"
- "A purchaser is either a one-time 'guest' OR a registered customer"
- "An invoice is either unpaid OR paid"

For these cases, we could use "OR" in our notes, like this:

```
data Email is UnvalidatedEmail OR ValidatedEmail
data ContactMethod is EmailAddress OR PhoneNumber
data Purchaser is GuestPurchaser OR RegisteredPurchaser (contains CustomerId)
data Invoice is UnpaidInvoice OR PaidInvoice
```

It's important to realize that these alternatives often play a critical role in business rules. For example:

- A password reset link should only be sent to a `ValidatedEmail` (never an `UnvalidatedEmail`).
- Discounts should only be given to a `RegisteredPurchaser` (never a `GuestPurchaser`).
- You can only apply a payment to an `UnpaidInvoice` (never a `PaidInvoice`).

Failing to clearly distinguish between these kinds of choices results in a confusing design at best, and possibly a seriously defective implementation.

Fighting the impulse to do class-driven design

One of the key tenets of Domain-Driven Design is *persistence ignorance*. It is an important principle because it forces you to focus on modeling the domain accurately, without worrying about the representation of the data in a database. Indeed, if you're an experienced object-oriented developer, then the idea of not being biased to a particular database model will be familiar, Object-oriented techniques such as dependency injection encourage you to separate the database implementation from the business logic.

But we also have to be careful of introducing bias into the design if we think in terms of objects and classes rather than the domain. For example, as the domain expert describes the different kinds of contact methods, you may be tempted to create a class hierarchy in your head, like this:

```
// represents all kinds of contact methods
class ContactMethodBase ...

// represents email contact method
class EmailAddressContactMethod extends ContactMethodBase ...

// represents phone contact method
class PhoneNumberContactMethod extends ContactMethodBase ...
```

But letting classes drive the design can be just as dangerous as letting a database drive the design — again, we're not really listening to the requirements.

- In our mental class hierarchy we have introduced an artificial base class, `ContactMethodBase`, that doesn't exist in the real world. This is a distortion of the domain. Try asking the domain expert what a `ContactMethodBase` is!
- And similarly, an `EmailAddress` is a reusable primitive value, not specific to contact methods, so to use it in this particular context we have had to create a wrapper class `EmailAddressContactMethod`. Again, this is another artificial object in the code that does not represent something in the domain.

The lesson here is that we should keep our minds open during requirements gathering and not impose our own technical ideas on the domain.

Workflows

So far, we've been talking about "nouns" in our domain — the data structures. In practice though, these are not the most important thing to model. Why is that?

Well, a business doesn't just *have* data, it *transforms* it somehow. That is, you can think of a typical business process as a series of data or document transformations. The value of the business is created in this process of transformation, so it is critically important to understand how these transformations work and how they relate to each other.

Static data — data that is just sitting there unused — is not contributing anything. So what causes a person (or automated process) to start working with that data and adding value? Often it's an outside trigger (a piece of mail arriving or your phone ringing), but it can also be a time-based trigger (you do something every day at 10 a.m.) or an observation (there are no more orders in the inbox to process, so do something else).

Whatever it is, it's important to capture it as part of the design. We generally call these things *Domain Events*. Domain Events are the starting point for almost all of the business processes we want to model. For example:

- "payment received" is a Domain Event that will kick off the "applying payment" workflow

- "email validation link clicked" is a Domain Event that will kick off the "validate email" workflow

There are number of ways to discover events in a domain, but one which is particularly suitable for a DDD approach is *Event Storming*, a collaborative process for discovering business events and their associated workflows which was developed by Alberto Brandolini.

So, given that we have discovered the events of interest, we next need to document the workflows (or use-cases) that are triggered by the events. To keep things at a high level, we will just document what the inputs and outputs are for each workflow, like this:

```
workflow ApplyPaymentToInvoice =
   triggered by: PaymentReceived
   inputs: UnpaidInvoice, Payment
   outputs: PaidInvoice

workflow ValidateEmail  =
   triggered by: EmailValidationLinkClicked
   inputs: UnvalidatedEmail, ValidationToken (from clicked link)
   outputs: ValidatedEmail OR an error
```

In the second case, the validation might fail (e.g. the link has expired) and so the output is written as a *choice* between alternatives: either a validated email address, or an error.

States and lifecycles

Most important business entities have a lifecycle — they go through a series of changes over time.

- An invoice starts off as unpaid, and then transitions to being paid.
- A purchaser starts off as a guest, and then transitions to being registered.

Even simple values can go through state transitions. For example, an email address starts off as unvalidated, and then transitions to being validated.

Being able to capture the states and transitions is an important part of domain modeling. To do this, we can use the same techniques described above: a set of choices to represent the various states, and workflows that transition between the states. For example, we can document the states and transitions for invoices like this:

```
// two states
data Invoice = UnpaidInvoice OR PaidInvoice

// one transition
workflow ApplyPaymentToInvoice =
    transforms UnpaidInvoice -> PaidInvoice
```

In this case, there is no way to transition from `PaidInvoice` to `UnpaidInvoice`. Should there be? Thinking in terms of states and lifecycles is a great way to trigger productive design discussions.

Understanding algebraic data types

Now that we've looked at some common concepts in domain modeling, let's look at how they can be mapped into code using algebraic data types. In this section, we'll define what *algebraic data types* are. And then in the next section, we'll see how they can be used to capture our domain model.

What are algebraic data types?

In an algebraic type system, new types are built from smaller types in two ways:

- By *AND*ing them together
- By *OR*ing them together

But what does *AND*ing and *OR*ing mean in the context of modeling?

"AND" Types

Let's start with building types using *AND*. For example, we might say that to make fruit salad you need an apple *AND* a banana *AND* some cherries. This kind of type is familiar to all programmers. It is a *record* or *struct*. Functional programmers call this a *product type*.

Here's how the definition of a `FruitSalad` record type would be written in F#:

```
type FruitSalad = {
  Apple : AppleVariety
  Banana : BananaVariety
  Cherries : CherryVariety
}
```

The curly braces indicate that it is a record type, and the three fields are `Apple`, `Banana`, and `Cherries`. The value in the `Apple` field must be of type `AppleVariety`, the `Banana` value must be of type `BananaVariety`, and so on.

"OR" Types

The other way of building new types is by using *OR*. For example, we might say that for a fruit snack you need an apple *OR* a banana *OR* some cherries: Functional programmers call this a *sum type* or *discriminated union*. I will call them *choice types* because they are used to represent choices in our domain.

Here's the definition of a `FruitSnack` using a choice type in F#:

```
type FruitSnack =
  | Apple of AppleVariety
  | Banana of BananaVariety
  | Cherries of CherryVariety
```

A vertical bar separates each choice, and the tags (such as `Apple` and `Banana`) are needed because sometimes the two or more choices may have the same type and so tags are needed to distinguish them. It can be read like this:

- A `FruitSnack` is either an `AppleVariety` (tagged with `Apple`) *OR* a `BananaVariety` (tagged with `Banana`) *OR* a `CherryVariety` (tagged with `Cherries`).

The varieties of fruit are themselves defined as choice types, which in this case is used similarly to an `enum` in other languages.

```
type AppleVariety =
  | GoldenDelicious
  | GrannySmith
  | Fuji

type BananaVariety =
  | Cavendish
  | GrosMichel
  | Manzano

type CherryVariety =
  | Montmorency
```

This can be read as:

- An `AppleVariety` is either a `GoldenDelicious` *OR* a `GrannySmith` *OR* a `Fuji`.
- and so on for the other types of fruit.

Unlike the `FruitSnack` example, there is no extra data associated with each case — they are just labels.

Simple types

We will often define a choice type with only *one* choice, such as this:

```
type EmailAddress =
  | EmailAddress of string
```

This definition is generally simplified to one line, like this:

```
type EmailAddress = EmailAddress of string
```

Why would we create such a type? Because it's an easy way to create a "wrapper" — a type that contains a primitive (such as a `string` or `int`) as an inner value.

For example, we might define wrapper types like these:

```
type PersonalName = PersonalName of string
type EmailAddress = EmailAddress of string
type PhoneNumber = PhoneNumber of string
```

and having done this, we can be sure that these three types cannot be mixed up or confused.

Function types

Finally, there's one more kind of type to discuss: function types. As we know from high school, a function is a kind of black box with an input and an output. Something goes in, is transformed somehow, and comes out the other side.

To describe a function abstractly, we just need to document the inputs and outputs. So for example, to document a function that adds 1 to a value, we would say that it takes an `int` as input and returns an `int` as output. It would be documented like this:

```
type Add1 = int -> int
```

where the type of the input is listed first, followed by an arrow, and then the type of the output.

But of course the inputs and outputs can be any type, including domain-specific types. Which means that we can document a function that creates an `EmailAddress` domain object from a string like this:

```
type CreateEmailAddress = string -> EmailAddress
```

This can be read as: to use `CreateEmailAddress` you need to provide a `string` as input and then the output is an `EmailAddress`.

Algebraic types are composable types

Now we can define what we mean by an algebraic type *system*. It's simply a type system where *every* compound type is composed from smaller types by *AND*-ing or *OR*-ing them together. Using *AND* and *OR* to build new data types should feel familiar — we used the same kind of *AND* and *OR* to document our domain earlier.

This kind of *composable* type system is a great aid in doing Domain-Driven Design because we can quickly create a complex model simply by mixing types together in different combinations, as we'll see next.

Modeling with algebraic data types

Now we have everything we need to do some real modeling. Let's revisit some of the domain descriptions at the top of this chapter, and model them using algebraic types.

Modeling order lines

The description was: "An order line has an order id, a product id, and an order quantity."

There are three distinct domain-specific primitive types, so we define these first:

```
type OrderId = OrderId of int
type ProductId = ProductId of int
type OrderQty = OrderQty of int
```

and then define a record that ANDs them together:

```
type OrderLine = {
  OrderId : OrderId
  ProductId : ProductId
  OrderQty : OrderQty
}
```

Modeling contact information

The description was: "Contact information consists of a personal name and a contact method, where a contact method is either an email address or a phone number."

Again, we have three "primitive" domain types:

```
type PersonalName = PersonalName of string
type EmailAddress = EmailAddress of string
type PhoneNumber = PhoneNumber of string
```

And then we can define a choice type with `EmailAddress` and `PhoneNumber` as alternatives.

```
type ContactMethod =
  | ByEmail of EmailAddress
  | ByPhone of PhoneNumber
```

Finally, we can combine the `PersonalName` and `ContactMethod` into a record:

```
type ContactInformation = {
  Name : PersonalName
  ContactMethod : ContactMethod
}
```

Modeling purchasers

The description was: "A purchaser is either a one-time 'guest' or has been registered on the website. A registered purchaser has been given a customer id, which a guest doesn't have."

We start by defining two distinct types, one for each case, plus a primitive `CustomerId` type:

```
type GuestPurchaser = {
  Name : PersonalName
  ContactMethod : ContactMethod
}

type CustomerId = CustomerId of int

type RegisteredPurchaser = {
  Id: CustomerId
  Name : PersonalName
  ContactMethod : ContactMethod
}
```

And we can then create a choice type with the alternatives:

```
type Purchaser =
  | Guest of GuestPurchaser
  | Registered of RegisteredPurchaser
```

Modeling email addresses

The description was: "An email is either validated or unvalidated. Password resets should only be sent to validated emails."

It's important to distinguish between `Unvalidated` and `Validated` emails — there are different business rules around them, so we should define distinct types for each of these, and a choice type to combine them:

```
type UnvalidatedEmailAddress = UnvalidatedEmailAddress of string
type ValidatedEmailAddress = ValidatedEmailAddress of string

type EmailAddress =
 | Unvalidated of UnvalidatedEmailAddress
 | Validated of ValidatedEmailAddress
```

We can document the password reset process like this:

```
type SendPasswordResetLink =
  ResetLinkUrl -> ValidatedEmailAddress -> output???
```

We're not sure what the output is, so we'll just leave it undefined for now. Also, details of how the email gets sent (e.g. via a SMTP server) are not relevant to the domain model right now, so we will not document that here.

Finally, we can document the email validation process like this:

```
type ValidationToken = ValidationToken of string // some opaque token

type ValidateEmailAddress =
  UnvalidatedEmailAddress -> ValidationToken -> ValidatedEmailAddress
```

which reads as: given an unvalidated email address and a validation token, we can create a validated email address.

Documenting failures

But that's not quite right, because we said that the action might fail. There was a *choice* between two kinds of output. If everything worked, we got a `ValidatedEmailAddress`, but if there was an error, some sort of error message. We can handle this by creating *another* choice type to represent the result:

```
type ValidationResult =
  | Ok of ValidatedEmailAddress
  | Error of string
```

and then we can use `ValidationResult` as the output of the validation process instead of `ValidatedEmailAddress`:

```
type ValidateEmailAddress =
  UnvalidatedEmailAddress -> ValidationToken -> ValidationResult
```

If we wanted to be specific about the kinds of errors that could occur, we could document them with yet another choice type, say `ValidationError` and then use that type in the result:

```
type ValidationError =
  | WrongEmailAddress
  | TokenExpired

type ValidationResult =
  | Ok of ValidatedEmailAddress
  | Error of ValidationError
```

This kind of "result" type is so common that it is available as a built-in generic type in most functional languages, defined like this:

```
type Result<'SuccessType,'FailureType> =
  | Ok of 'SuccessType
  | Error of 'FailureType
```

Using this built-in type then, the function definition looks like this:

```
type ValidateEmailAddress =
  UnvalidatedEmailAddress -> ValidationToken -> Result<ValidatedEmailAddress,Validat\
ionError>
```

which clearly shows us that the validation might fail, and what kinds of errors we can expect. Furthermore, since this is *code* not documentation, we can be sure that any implementation *must* match this design exactly.

Sketching a domain model by composing types

This kind of modeling is not a heavyweight "big design up front". Just the opposite. It's very useful for "design sketches", written in conjunction with a domain expert during a discussion.

For example, say that we want to track payments for an e-commerce site. Let's see how this might be sketched out in code during a design session.

First, we start with some wrappers for the primitive types, such as `CheckNumber`. These are the "primitive types" we discussed above. Doing this gives them meaningful names and makes the rest of the domain easier to understand.

```
type CheckNumber = CheckNumber of int
type CardNumber = CardNumber of string
```

Next, as we discuss credit cards further, we might build up some more low-level types. A `CardType` is an *OR* type — a choice between `Visa` *or* `Mastercard`, while `CreditCardInfo` is an *AND* type, a record containing a `CardType` *and* a `CardNumber`:

```
type CardType = Visa | Mastercard

type CreditCardInfo = {
  CardType : CardType
  CardNumber : CardNumber
}
```

We learn that the business will accept cash, checks, or credit cards, so we then define another *OR* type, `PaymentMethod`, as a choice between `Cash` or `Check` or `Card`. This is no longer a simple "enum" because some of the choices have data associated with them: the `Check` case has a `CheckNumber` and the `Card` case has `CreditCardInfo`:

```
type PaymentMethod =
  | Cash
  | Check of CheckNumber
  | Card of CreditCardInfo
```

Next we might talk about money, which leads us to define a few more types, such as `PaymentAmount` and `Currency`:

```
type PaymentAmount = PaymentAmount of decimal  // must be positive
type Currency = EUR | USD
```

And finally, the top-level type, `Payment`, is a record containing a `PaymentAmount` *and* a `Currency` *and* a `PaymentMethod`:

```
type Payment = {
  Amount : PaymentAmount
  Currency: Currency
  Method: PaymentMethod
}
```

So there you go. In about 25 lines of code, we have modeled the domain and defined a pretty useful set of types that can guide the implementation.

Of course, there is no behavior directly associated with these types because this is a functional model, not an object-oriented model. To document the actions that can be taken, we instead define types that represent functions.

So, for example, if we want to show there is a way to use a `Payment` type to pay for an unpaid invoice, where the final result is a paid invoice, we could define a function type that looks like this:

```
type PayInvoice = UnpaidInvoice -> Payment -> PaidInvoice
```

Which can be read as: given an `UnpaidInvoice` and then a `Payment`, we can create a `PaidInvoice`.

Or, to convert a payment from one currency to another:

```
type ConvertPaymentCurrency = Payment -> Currency -> Payment
```

where the first `Payment` is the input, the second parameter (`Currency`) is the currency to convert to, and the second `Payment` — the output–is the result after the conversion.

Value Objects, Entities and Aggregates

We've now got a basic understanding of how to model the domain types and workflows, so let's move on and look at an important way of classifying domain objects, based on whether they have a persistent identity or not. In DDD terminology, objects with a persistent identity are called *Entities* and objects without a persistent identity are called *Value Objects*.

Value objects

In many cases, the data objects we are dealing with have no identity — they are interchangeable. For example, one instance of a `CustomerId` with value "1234" is the same as any other `CustomerId` with value "1234." We do not need to keep track of which one is which — they are equal to each other.

When we model a domain using an algebraic type system, the types we create will implement this kind of field-based equality testing automatically. We don't need to write any special equality code ourselves, which is nice. To be precise, in an algebraic type system, two record values (of the same type) are equal if all their fields are equal, and two choice types are equal if they have the same choice case, and the data associated with that case is also equal. This is called *Structural Equality*.

Entities

However, we often model things that, in the real world, do have a unique identity, even as their components change. For example, even if I change my name or my address, I am still the same person. In DDD terminology, we call such things *Entities*.

In a business context, Entities are often a document of some kind: Orders, Quotes, Invoices, Customer profiles, product sheets, etc. They have a *lifecycle* and are transformed from one state to another by various business processes.

Entities need to have a stable identity despite any changes, and therefore, when modeling them, we need to give them a unique identifier or key, such as an "Order Id," or "Customer Id." For example, the `UnpaidInvoice` type below has an `InvoiceId` that stays the same even if the `AmountDue` changes.

```
type UnpaidInvoice = {
  Id: InvoiceId
  AmountDue : InvoiceAmount
}
```

And of course, when we model different states in a lifecycle with distinct types, we must ensure that all the states have a common identifier field. That means, for example, that a `PaidInvoice` must have the same `InvoiceId` as the corresponding `UnpaidInvoice`:

```
type PaidInvoice = {
  Id: InvoiceId
  AmountPaid : InvoiceAmount
  PaymentDetails :  PaymentDetails
}
```

We saw earlier that, by default, equality in an algebraic type system uses all the fields of a record. But when we compare Entities we want to use only one field, the identifier. So in order to model Entities correctly we must change the default behavior.

One way of doing this is to create a custom equality test so that only the identifier is used, but that can be error prone in some cases. Therefore, a sometimes preferable alternative is to disallow equality testing on the object altogether!

In F# this can be done by adding a `NoEquality` type annotation like this:

```
[<NoEquality; NoComparison>]
type UnpaidInvoice = {
  Id: InvoiceId
  AmountDue : InvoiceAmount
}
```

Now when we attempt to compare values with this annotation, we get a compiler error. Of course we can still compare the `InvoiceId` fields directly. The benefit of the "NoEquality" approach is that it removes any ambiguity about what equality means for a particular type, and forces us to be explicit.

Immutability as a design aid

In functional programming languages, algebraic data types are immutable by default, which means that none of the objects defined so far can be changed after being initialized.

How does this affect our design?

- For *Value Objects*, immutability is required. Think of how we use them in common speech: if we change any part of a personal name, say, we call it a *new*, distinct name, not the same name with different data. The fact that immutability is the default means that Value Objects are very easy to implement — no extra work is needed.
- For *Entities*, it's a different matter. We expect the data associated with Entities to change over time; that's the whole point of having a constant identifier. So how can immutable data structures be made to work this way? The answer is that we make a *copy* of the Entity with the changed data while preserving the identity. All this copying seems like it might be a lot of extra work but isn't an issue in practice. Functional programming languages have built in support to make this easy.

A benefit of using immutable data structures is that any changes have to be made *explicit* in the type signature. For example, if we want to write a function that changes the `AmountDue` field in a `UnpaidInvoice`, we can't use a function with a signature like this (where `unit` means no output):

```
type UpdateAmountDue = UnpaidInvoice -> AmountDue -> unit
```

That function has no output, which implies that nothing changed! Instead, our function must have a `UnpaidInvoice` type as the output as well, like this:

```
type UpdateAmountDue = UnpaidInvoice -> AmountDue -> UnpaidInvoice
```

This clearly indicates that, given a `UnpaidInvoice` and a `AmountDue`, a different `UnpaidInvoice` is being returned. Immutability has forced us to make the logic explicit in the design.

Immutability and aggregate boundaries

Immutability can be particularly helpful for determining aggregate boundaries. Say that we have an `Order` that contains a list of `OrderLines`:

```
type OrderLine = {
  OrderLineId : OrderLineId
  ProductId : ProductId
  OrderQty : OrderQty
}

type Order = {
  OrderId : OrderId
  Lines : OrderLine list
}
```

But now here's a question: if you change an `OrderLine`, have you also changed the `Order` that it belongs to? In this case, it's clear that the answer is yes: changing a line also changes the entire order.

With immutable data. this design is unavoidable. If I have an immutable `Order` containing immutable `OrderLines`, then just making a copy of one of the order lines *does not* also make a copy of the `Order` as well. In order to make a change to an `OrderLine` contained in an `Order`, I need to make the change at the level of the `Order`, not at the level of the `OrderLine`.

For example, here's the definition of a function that updates the price of a specific order line. We need an `Order`, the `OrderLineId` of the line to change, and the new `Price`:

```
type ChangeOrderLinePrice = Order -> OrderLineId -> Price -> Order
```

The final result, the output of the function, is a new `Order` containing a new list of lines, where one of the lines has a new price. You can see that immutability causes a ripple effect in a data structure, whereby changing one low-level component can force changes to higher-level components too.

Therefore, even though we're just changing one of its "subentities" (an `OrderLine`), we *always* have to work at the level of the `Order` itself. In other words, immutability has forced us to define the aggregate root to be the `Order`, and requires us to make all changes at this level rather than at the order line level, which is just what we want.

Aggregate references

Now let's say that we need information about a customer to be associated with an `Order`. A bad design might be to add the `Customer` as a field of an `Order`, like this:

```
type Order = {
  OrderId : OrderId
  Customer : Customer // info about associated customer
  Lines : OrderLine list
  // etc
  }
```

How can we tell that this is a bad design? The ripple effect of immutability forces us to say that if I change any part of the *customer*, I must also change the *order* as well. Is that really what we want?

No, probably not. It's clear that a better design would be to store a *reference* to the customer, not the whole customer record itself. That is, we would just store a `CustomerId` in the `Order` type, like this:

```
type Order = {
  OrderId : OrderId
  CustomerId : CustomerId // reference to associated customer
  Lines : OrderLine list
  // etc
}
```

Then when we need the full information about the customer, we would get the `CustomerId` from the `Order` and then load the relevant customer data from the database separately, rather than loading it as part of the order. In other words, the `Customer` and the `Order` are *distinct* and *independent* aggregates. They each are responsible for their own internal consistency, and the only connection between them is via the identifiers of their root objects.

Again, we can see that immutability has been used as a design aid to help us discover whether entities are part of the same aggregate (orders and order lines) or are separate (orders and customers).

Making illegal states unrepresentable

Since we've gone to this trouble to model the domain properly, we should take some precautions to make sure that any data in this domain is valid and consistent. The goal is to create a bounded context that always contains data we can trust, with a distinct boundary from the untrusted outside world. If we can be sure that all data is always valid, the implementation can stay clean and we can avoid having to do defensive coding.

So, in this final section, we'll look at some techniques to avoid invalid data by making illegal states unrepresentable. That is, the rules and logic are made explicit in the *design itself*, rather than relying on checks buried in the code somewhere. Furthermore, in a type-checked language, the data cannot become invalid because the compiler will not allow it! This means that fewer unit tests are needed, less defensive code, and less implementation effort in general.

Here as some common things that we can do in the design:

- Avoid nulls with optional values
- Enforce constraints
- Enforce business rules

Avoiding nulls with optional values

Nulls have been the bane of programmers for 50 years. Their own creator called them a "billion dollar mistake". From a design point of view, they complicate matters because we can never be sure whether a value is meant to be optional or not.

In most languages with algebraic type systems, nulls do not exist. That means that every time we reference a value in a domain model, it's a *required* value. But of course we *do* sometimes need to indicate that data might be optional in certain cases. How can we represent that in the design?

The answer is to think about what missing data means: it's either present or absent. There's something there, or nothing there. We can model this with a special choice type (called `Option` in F# and `Maybe` in other languages), defined like this:

```
type Option<'a> =
  | Some of 'a
  | None
```

The `Some` case means that there is data stored in the associated value `'a`. The `None` case means there is no data. The tick in `'a` is F#'s way of indicating a generic type — that is, the `Option` type can be used to wrap *any* other type. The C# or Java equivalent would be something like `Option<T>`.

To indicate optional data in the domain model then, we wrap the type in `Option<...>` just as we would in C# or Java. For example, if we have a `PersonalName` type and the first and last names are required, but the middle initial was optional, we could model it like this:

```
type PersonalName = {
  FirstName : string            // required
  MiddleInitial: Option<string> // optional
  LastName : string             // required
}
```

This code makes it very clear which fields are optional and which are required.

Enforcing constraints

It is very rare to have a unbounded integer or string in a real-world domain. Almost always, these values are constrained in some way. In the earlier examples we mentioned some values like this:

```
data OrderQuantity is int   // Constrained to be between 1 and 100.
data EmailAddress is string // Must not be null or empty. Must contain @ symbol.
```

Rather than allowing the raw `int` and `string` values to be used, we want to ensure that, if we *do* have a `OrderQuantity` or `EmailAddress`, then we know *for sure* that the constraints have been satisfied. If we never need to check the constraints after creation, then that eliminates yet more defensive coding.

This sounds great, so how do we ensure that the constraints are enforced? The answer: the same way as we would in any programming language — make the constructor private and have a separate function that creates valid values and rejects invalid values, returning an error instead. In FP communities, this is sometimes called the *smart constructor* approach.

Here's an example of this approach applied to `OrderQuantity`:

```
type OrderQuantity = private OrderQuantity of int
                  // ^ private constructor

// Define a module with the same name as the type
module OrderQuantity =

  /// Define a "smart constructor" for OrderQuantity
  let create qty =
    if qty < 1 then
      // failure
      Error "OrderQuantity can not be negative"
    else if qty > 100 then
      // failure
      Error "OrderQuantity can not be more than 100"
    else
      // success -- construct the OrderQuantity value and return it
      Ok (OrderQuantity qty)
```

So now an `OrderQuantity` value can *not* be created directly, due to the private constructor. But it *can* be created using the "factory" function `OrderQuantity.create`.

The `create` function accepts an `int` and returns a `Result` type to indicate a success or failure. These two possibilities are made explicit in its function signature:

```
int -> Result<OrderQuantity,string>.
```

The design tells us straight away that creating an `OrderQuantity` might fail. But, once created, immutability ensures that an `OrderQuantity` will never change and thus always be valid thereafter.

Capturing business rules in the design

Can we document business rules using just the type system? That is, we'd like to use the type system to represent what is valid or invalid so that the compiler can check it for us, instead of relying on runtime checks or code comments to ensure the rules are maintained.

Surprising often, we *can* do this. To see this approach in action, let's say we have a business rule around contacting a customer: "A customer must have an email or a postal address."

How should we represent this? We might start by creating a record with both an `Email` and an `Address` property, like this:

```
type Contact = {
  Name: Name
  Email: EmailContactInfo
  Address: PostalContactInfo
}
```

But this is an incorrect design. It implies both `Email` and `Address` are required.

OK, so let's make them optional:

```
type Contact = {
  Name: Name
  Email: Option<EmailContactInfo>
  Address: Option<PostalContactInfo>
}
```

But this is not correct either. As it stands, `Email` and `Address` could both be missing, and that would break the business rule. Now, of course, we could add special runtime validation checks to make sure that this couldn't happen. But can we do better and represent this in the type system? Yes, we can!

The trick is to look at the rule closely. It implies that a customer:

- Has an email address only, or
- Has a postal address only, or
- Has both an email address and a postal address.

That's only *three* possibilities. And how can we represent these three possibilities? With a choice type of course!

```
type BothContactMethods = {
  Email: EmailContactInfo
  Address : PostalContactInfo
}

type ContactInfo =
  | EmailOnly of EmailContactInfo
  | AddrOnly of PostalContactInfo
  | EmailAndAddr of BothContactMethods
```

And then we can use this choice type in the main `Contact` type, like this:

```
type Contact = {
  Name: Name
  ContactInfo : ContactInfo
}
```

Again what we've done is good for developers (we can't accidentally have no contact information — one less test to write) but also it is good for the *design*. The design makes it very clear that there are only three possible cases, and exactly what those three cases are. We don't need to look at documentation, we can just look at the code itself.

Conclusion

If we look at some of the snippets above, we should be pleased. We have managed to capture the essence of the verbal domain descriptions, but as code rather than as text or documentation.

From a developer's point of view, we have attained one of the goals of Domain-Driven Design — compilable code that defines and implements the domain model and ubiquitous language. Another developer joining the project would be able to understand the domain model without having to translate between the implementation concepts and the domain concepts.

Furthermore, this code is still quite comprehensible for non-developers. With a little training, I believe it would be possible for a domain expert to understand this code — probably no harder than understanding UML digrams or other kinds of technical documentation. It certainly is more readable than a conventional programming language such as C# or Java.

This approach, using types as documentation, is very general and it should be clear now how we can apply it to almost any domain modeling situation. Because there's no implementation at this point, it's a great way to try ideas out quickly when you are collaborating with domain experts. And of course, because it is just text, a domain expert can review it easily without needing special tools, and maybe even write some types themselves!

Domain Modeling with Monoids — Cyrille Martraire

It is unlucky the word “Monoid” is so awkward, as it represents a concept that is both ubiquitous and actually simple.

Monoids are all about *composability*. It’s about having abstractions in the small that compose infinitely in the large.

You are already familiar with many cases of typical monoidal composeablity, such as everything *group-by*, or everything *reduce*. What I want to emphasize is that Monoids happen to be frequent occurrences in business domains, and that you should spot them and exploit them.

Since I’ve first talked about my enthusiasm for monoids at conference around the world, I’ve received multiple positive feedbacks of concrete situations where monoids helped teams redesign complicated parts of their domain models into something much simpler. And they would then say that the new design is “more elegant”, with a smile.

Just one important warning: please don’t confuse monoids with monads. Monoids are much easier to understand than monads. For the rest of the text, we will keep monads out of scope, to focus solely on monoids and their friends.

What are Monoids?

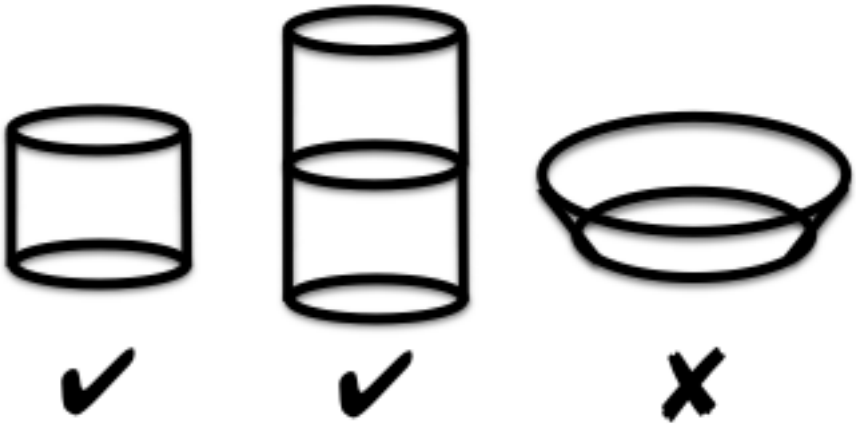
Monoids come from a part of mathematics called abstract algebra. It’s totally intimidating, but it really doesn’t have to, at least not for monoids, which are creatures so simple that kids pretty much master them by age of 2 (without knowing of course)

So what are monoids? A monoid is a mathematical structure. First we start with a simple set. A set is just a criterion to decide if elements belong or not to the set. Yes it’s a bit circular a definition but you get the idea.

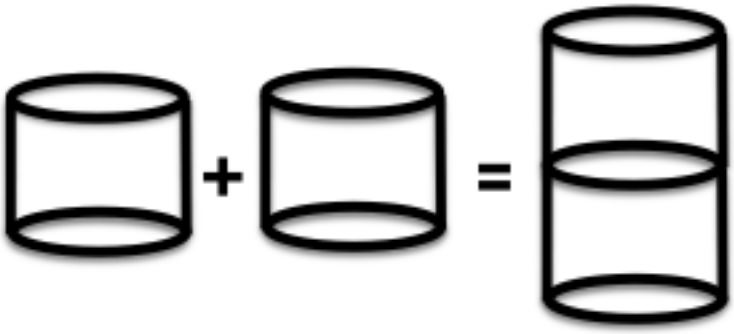
I usually explain monoids through glasses of beer, but here I will use plumbing pipes. For example, let’s define a set of pipes this way:

The set of pipes with a hole this size

Given this definition of our set, now we can test if various elements belong to it or not.



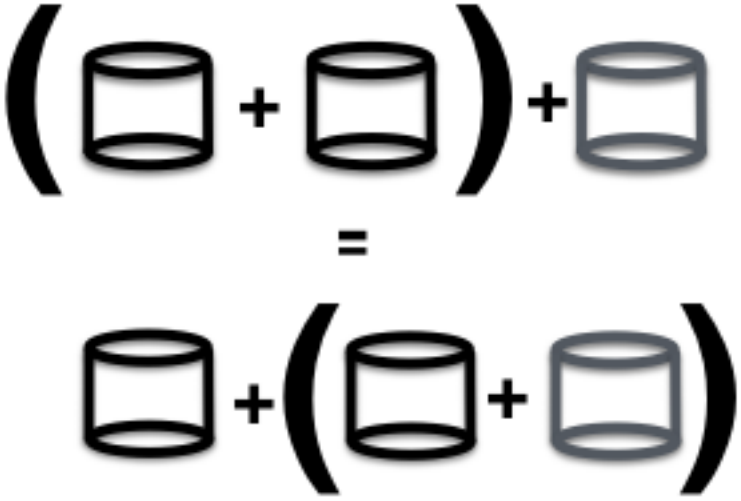
On top of this set, we define an operation, that we can call “combine”, “append”, “merge”, or “add”, that takes two elements and combines them.



One funny thing we notice now is that given two elements from the set, the result is always… in the

set too! That sounds like nothing, but that's what Closure of Operations is all about: the set is closed under this operation. It matters. We have a little system that's all about itself, always. That's cool.

But there's more to it. If you first combine the first two pipes together, then combine the result with the third pipe, or if you first combine the last two pipes then you combine the first pipe with it, then you end up with the exact same result. We're not talking about changing the ordering of the pipes, just the ordering of combining them. It's like putting the parenthesis anywhere doesn't change the result. This is called Associativity, and is important.

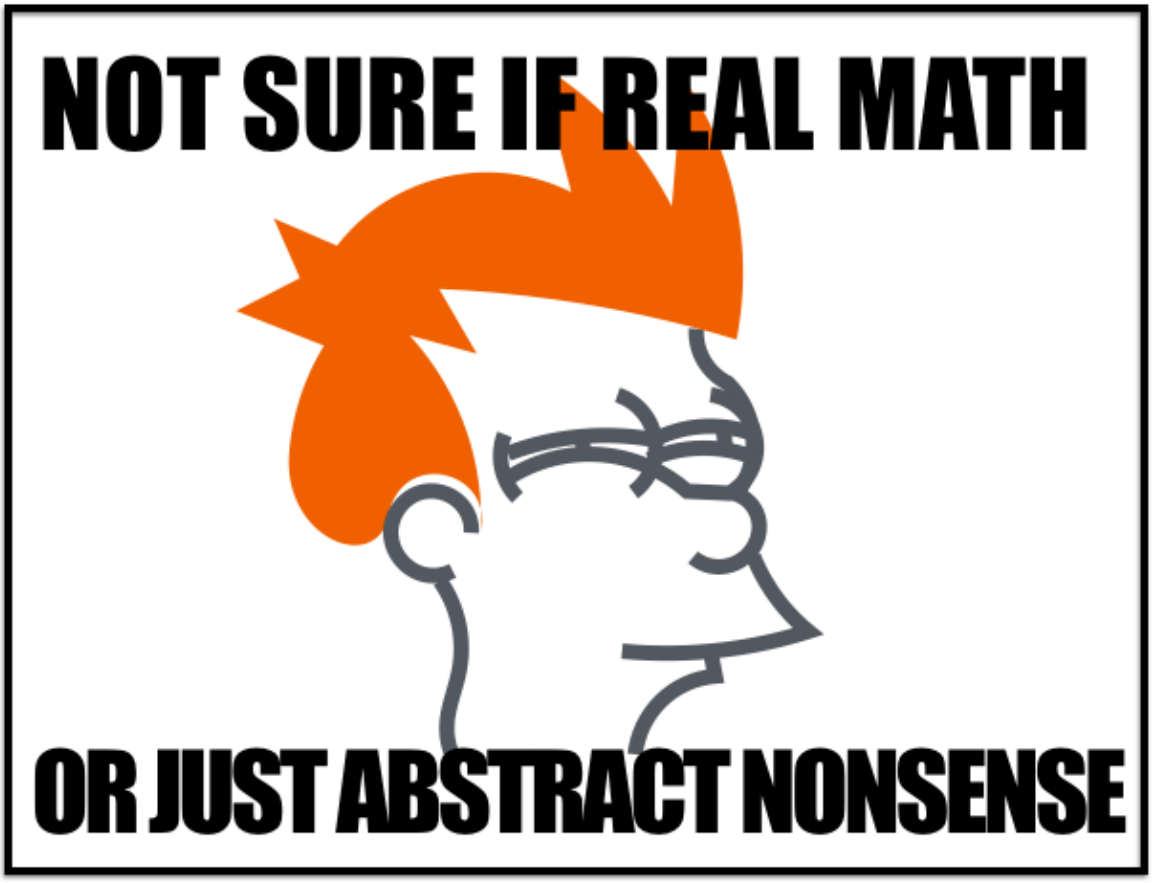


By the way changing the ordering of the pipes would be called Commutativity, and we have it if they all are identical, but this property is not as frequent.

So we have a set of elements, an operation with results that are all in the set, and that is associative. To really be a monoid, you need one more thing: you have to define one more kind of pipe, that is invisible, so I can't show it to you. But believe me, it exists, because as a developer I can create the system the way I prefer it to be. This special element belongs to the set: it has a hole the right size. So whenever I combine this special element with any other, the result is just the other element itself. It doesn't change anything. That's why we call it the neutral, or identity element.

And voilà! A set, an operation, closure of this operation, associativity, and neutral element: that's the formal definition of a monoid!

Ok at this point, you are perhaps like:



But stay with me, we'll see how this really closely relates to your daily job.

So what?

So when I started explaining monoids to people I also explained it to my wife. She instantly got it, but then asked: what for?

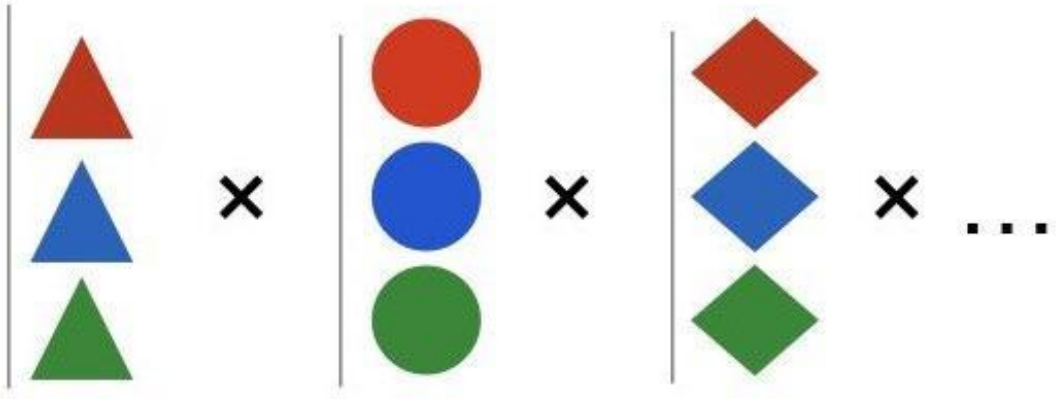
So I was the one wondering for a few seconds. Why does it matter to me? I believe it's all about dealing with encapsulating some diversity inside the structure.

You probably know the old joke in programming:

> There are only three numbers in programming: 0, 1, and MANY.

That's so true. But this also illustrates a very common kind of diversity we face all the time: the singular, the plural, and the absence. Monoids naturally represent the singular, with the elements of the set. It also deals with the plural, thanks to the *combine* operation that can take a plural and turn it back into an element as usual. And the neutral element takes care of the absence case, and it also belongs to the set. So monoids encapsulate this diversity inside their structure, so that from the outside you don't have to care about it. That's a great idea to fight the battle against complexity.

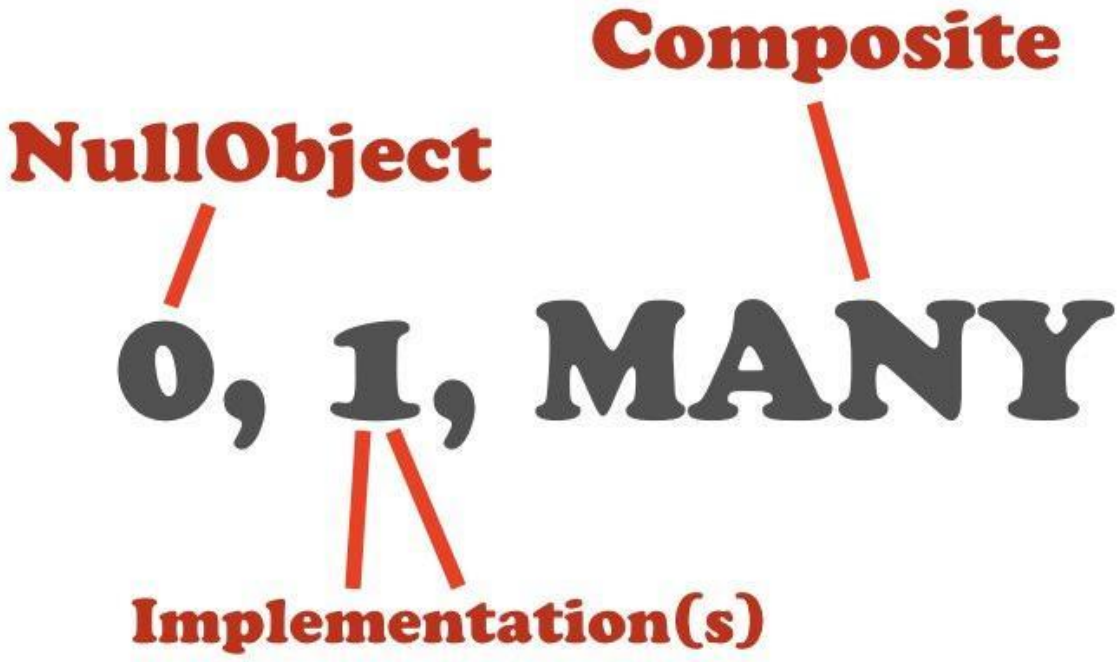
In the wild real life problems that we have, if we're not careful, we have to deal with various concepts, each potentially having their own diversity of being either singular, plural, or nothing. In the worst case, you'd end up with the cartesian product of all cases. It doesn't scale well.



Once you encapsulate this diversity inside a monoid for each concept, then you only have one case

for each, and together it remains one single case.

If you apply that often, then you can deal with high levels of complexity with ease. Monoids help scale in complexity. In fact, if you're comfortable in object-oriented programming, you may recognize something familiar you're doing already: for a given interface, I often find myself using the Composite pattern to deal with the plural, and the NullObject pattern to deal with the absence. These patterns help deal with singular, plural and absence consistently, so that the caller code doesn't even have to know. That's similar in purpose.



Examples Please!

You already know a lot of monoids in your programming language:

- **Integer** with **addition**: Integers are closed under addition: `int + int = int`. They're associative: `(3+5)+2=3+(5+2)`, and their neutral element is `0`, since any integer plus zero gives the same integer.
- **Lists** with list **append** operation: `List + List = List` is closed under this appending, which is associative: `(a)+(b,c)=(a, b)+(c)`. And the empty list is the neutral element here.
- A special case of lists, **Strings** with **concatenation**: `"hello" + "world"` is a string too. It's associative: `"cy"+"ri"+"lle"`, and the neutral element is the empty string.

Note that integers can also form a monoid with the multiplication operation, in which case the neutral element would be `1`. But natural integers with subtraction do not form a monoid, because `3 - 5` is not in the set of natural integers.

All this is not difficult. Still, such a simple thing is a key to very complex behaviors. It's also the key to infinite scalability of space (think Hadoop), and the key to infinite incremental scalability (think Storm). There's one joke in Big Data circles:

> If you're doing Big Data and you don't know what an abelian group is, then you do it wrong!

It's all about *composability*, which is highly desirable pretty much everywhere. .

Implementing monoids in your usual programming language

So how do we implement monoids in plan Java code?

> Monoids are typical Functional Programming; In Functional Programming everything is a value; Therefore: Monoids are values!

That's a solid proof that monoid are value objects. Seriously though, they do have to be value objects, i.e. immutable and equality by value. But monoid objects don't have to be anemic, with just data. They are supposed to have behavior, and in particular behavior that compose, like lengths, where we want to be able to write: `18 m + 16 m = 34 m`. The corresponding code for this method would be:

```
public Length add(Length other){
   return new Length(value + other.value);
}
```

This add() method returns a new instance, as value objects should do. It must not perform any side-effect, as advocated by the DDD "Side-Effect-Free Functions" pattern. Immutability and Side-Effect-Free Functions together are good taste! That should be your default style of programming, unless you really have to do otherwise.

In addition, being immutable and side-effect-free means that testing is a no-brainer: just pass data in, and assert the result out. Nothing else can happen to make it more complicated.

Monoids in domain modeling

Domain-specific lists, like a mailing list defined as a list of emails addresses, can form a monoid at least twice, once with the union operation, and a second time with the intersection operation. The neutral element would be *nobody()* in the former case, and *everybody()* in the latter case. Note the naming of the neutral elements that is domain-specific, instead of more generic names like *empty* or *all*. But we could go further and rename the intersection() operation into a word like *overlapping* is this was the way the domain experts talked about this problem.

Money and Quantity

Let's start with the good old Money analysis pattern, from Martin Fowler: `(EUR, 25) + (EUR, 30) = (EUR, 55)`. And in case the currencies don't match, we would throw an exception in the add method. Note that throwing exception indeed break perfect composability, but in practice we could deal with some, as long as they only reveal coding mistakes.

Our money class would be defined in UML like this:

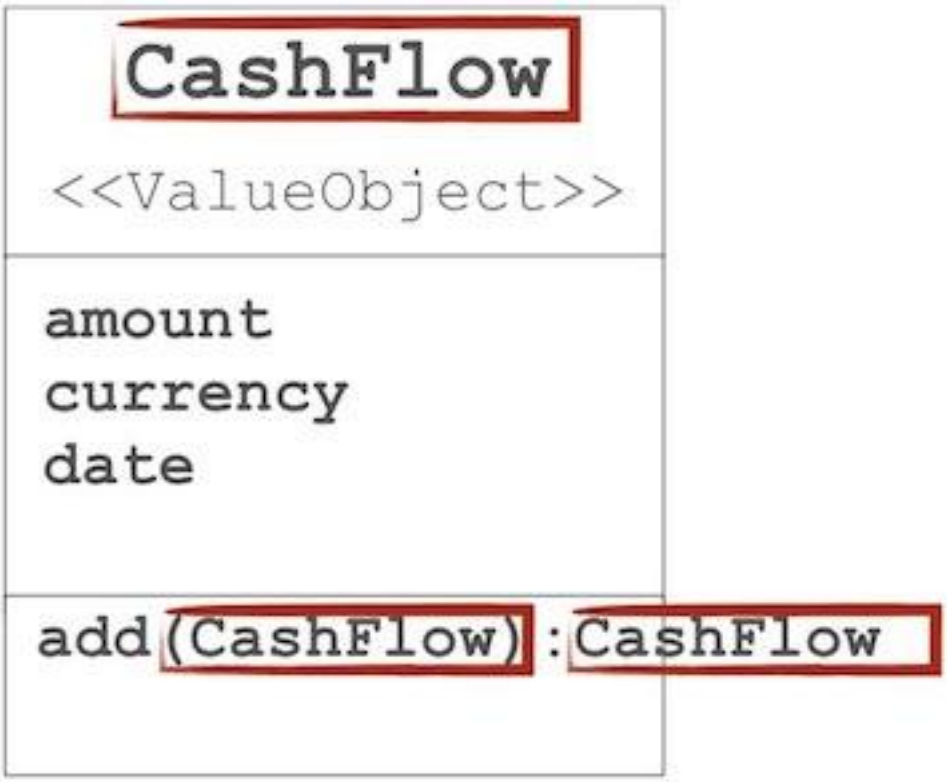
The Money pattern is indeed a special case of the more general Quantity analysis pattern: "Representing dimensioned values with both their amount and their unit" (Fowler).

Cashflows and sequences of cashflows

Now that we have a money amount, we can make it into a cashflow, by adding a date:

```
  (EUR, 25, TODAY)
+ (EUR, 30, TODAY)
= (EUR, 55, TODAY)
```

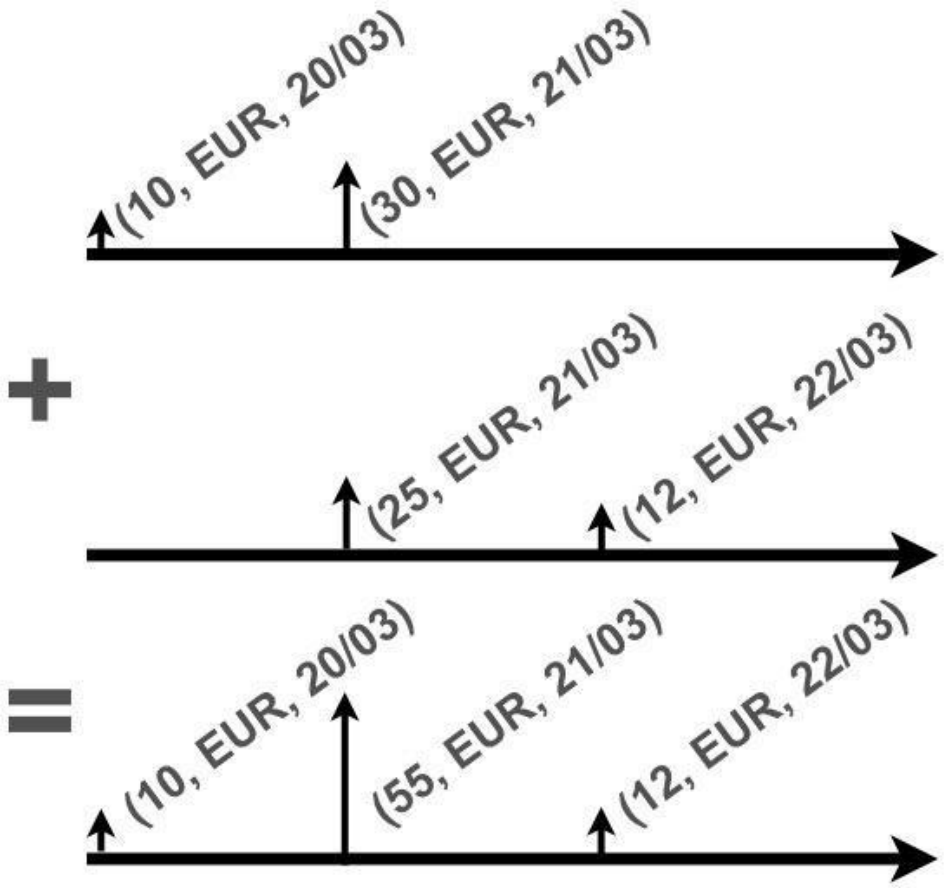
And again we could throw an exception if the dates don't match.



Looking at the UML class diagram, it's striking that the class only refers to its own type, or primitives (in the constructor, not shown here). Methods take Cashflow as parameter, and return Cashflow, and nothing else.

This is what Closure of Operation means in code. This type is egotistic, it only talks about itself. That's a desirable quality for code, as advocated in the DDD book, and it's one of the mandatory properties of a monoid.

But why stop there? We typically deal with many cashflows that go together, and we also think of them as stuff we can add:



So once again, we want to be able to write the code the exact same way:

```
  Cashflow Sequence
+ Cashflow Sequence
= Cashflow Sequence
```

At this point you get the picture: monoid are all about what we could call an arithmetics of objects.

Note that the addition operation in the Cashflow Sequences above is in basically the list concatenation (e.g. `addAll()` in Java), where cashflows on the same date and on the same currency are then added together using the addition operation of the cashflow themselves.

Ranges

A range of numbers or a range of dates can be seen as a monoid, for example with the compact-union operation, and the empty range as the neutral element:

```
[1, 3] Union [2, 4] = [1, 4] // compact union
[1, 3] Union ][ = [1, 3]     // neutral element
```

By defining the operation of "compact" union:

```
public final class Range
{
   private final int min;
   private final int max;
   public final static EMPTY = new Range();

   public Range union(Range other)
   {
       return new Range(
          min(this.min, other.min),
          max(this.max,other.max)
       );
   }
}
```

Note that the internal implementation can absolutely delegate the work to some off-the-shelf implementation, e.g. some well-known, well-tested open-source library.

Predicates

Predicates are natural monoids, with logical `AND` and the `ALWAYS_TRUE` predicate, or with logical `OR` and the `ALWAYS_FALSE` predicate.

Grants

But even unexpected stuff like read/write/execute grants can form a monoid with some merge operation defined for example as "the most secure wins":

```
r merge w = r
w merge x = w
```

The implementation could be an enum and perform a `MIN` on the internal ordering of each value.

```
public final enum Grant
{
  R, W, X;

  public Grant merge(Grant other)
  {
     return
       this.ordinal() < other.ordinal() ? this : other;
  }
}
```

Of course it's up to your domain expert to decide which exact behavior is expected here, and how the operation should be named.

Monoids of monoids are monoids

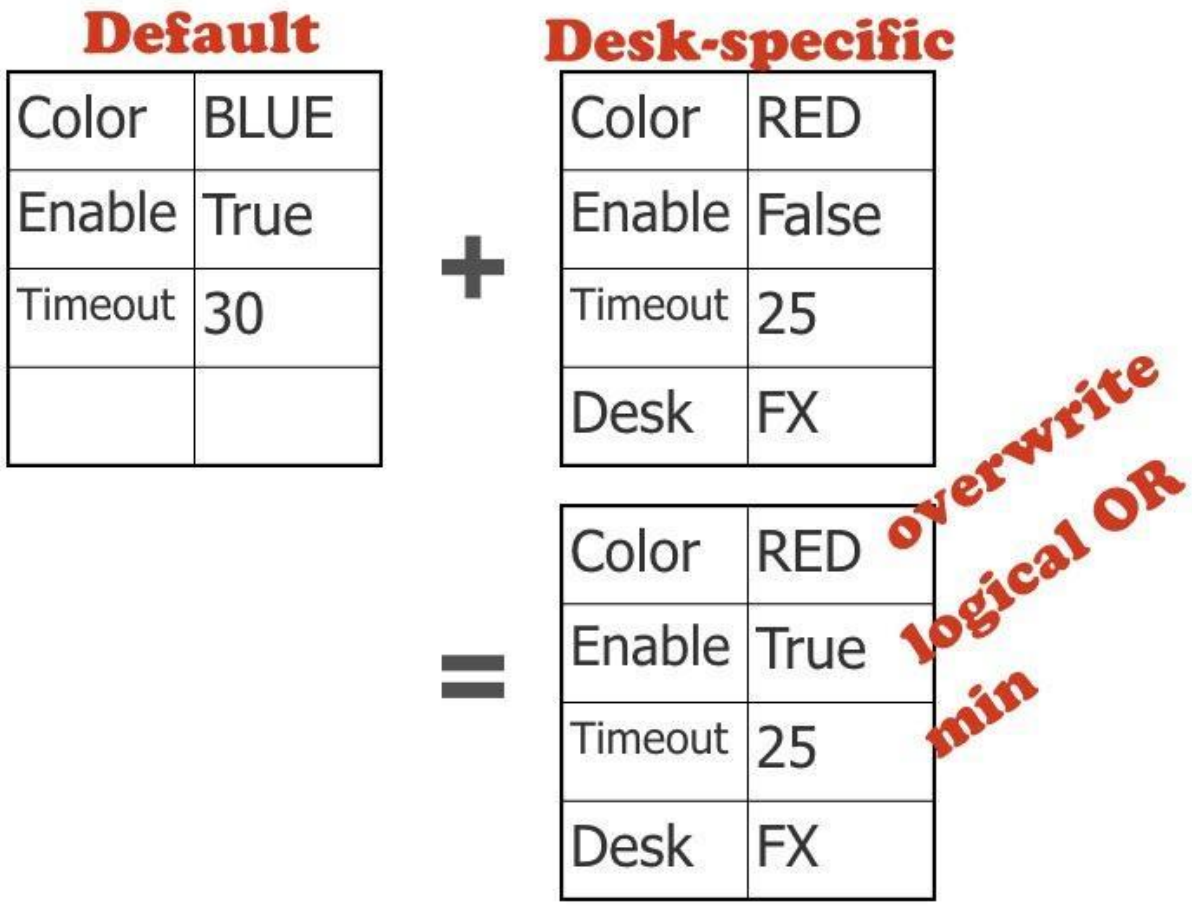
Nesting monoids can easily lead to monoids. For example in many systems you have configuration maps for the settings of an application. You often have a default hardcoded one, then by order of precedence one by department, then another by desk, and ultimately one by user. This leads naturally to a monoid form:

```
MonoidMap + MonoidMap = MonoidMap
```

One simple way to do that is just combine the maps with the `LAST ONE WINS` policy:

```
public MonoidMap append(MonoidMap other)
{

  Map<String, Object> result = new HashMap<>(this.map);
  result.putAll(other.map);
  return new MonoidMap(result);
}
```

But we can go further if all values are also monoids, and let each value make its own monoidal magic:



In our example, colors are combined by an `OVERWRITE` operation (last value wins), Enable values are combined by a logical `OR` operation, while Timeout values are combined by an integer `MIN` operation. You can see here that all the value are monoids by themselves with these operations. By defining the map-level combine operation (here noted +) by delegating to the monoid operation of each value, in parallel for each key, then we also have the configuration maps as monoids. Their neutral element could be either an empty map, or a map with all the neutral elements of each type of value.

```
public NestedMonoidMap append(NestedMonoidMap other)
{
  Map<String, Monoid<?>> result = new HashMap<>(map);
  for (String key:other.map.keySet()){
    Monoid value = map.get(key);
    Monoid value2 = other.map.get(key);

    result.put(key, value == null ? value2 : value.append(value2));
  }

  return new NestedMonoidMap(result);
```

```
}
```

Of course in this example, each value would have to be itself a monoid, with its own specific way to append or merge.

What I like in this example is also that it shows that value objects don't have to be small-ish. We can have huge objects trees as values and as monoids, and it works well. And don't obsess too much about the memory allocation here, most of the values are reused many times, really.

Non Linear

But not everything is that easy to model as monoids. For example, if you have to deal with partial averages and want to compose them into a bigger average, you cannot write: `Average + Average` as it would just be `WRONG`:

```
Average + Average = WRONG
```

Average calculation just doesn't compose at all. This makes my panda sad.

But if you really want to make it into a monoid, then you can do it! The usual trick is to go back to the intermediate calculation, in which you can find some composable intermediate sub-calculations:

```
avg = sum / count
```

And it turns out that put together as a tupple, it composes quite well, using a tuple-level addition defined as the addition of each term:

```
(sum, count) + (sum, count) = (sum, count)
```

Which internally becomes:

```
  (sum_0, count_0)
+ (sum_1, count_1)
= (sum_0 + sum_1, count_0 + count_1)
```

So you can combine tuples at large scale, across many nodes for example, and then when you get the final result as a tuple, then you just finish the work by taking the average out of it by actually doing the division sum/count.

```
public class Average
{
  private final int count;
  private final int sum;

  public static final Average NEUTRAL = new Average(0, 0);

  public static final Average of(int... values)
  {
    return new Average( values.length, stream(values).sum());
  }

  private Average(int count, int sum)
  {
    this.count = count;
    this.sum = sum;
  }

  public double average()
  {
    return (double) sum / count;
  }

  public int count()
  {
    return count;
  }

  public Average add(Average other)
  {
    return new Average( count + other.count, sum + other.sum);
  }

  // hashcode, equals, toString

}
```

And if you need the standard deviation, you can do the same trick, just by adding the sum of the values at the power of two (`sum2`):

```
  (sum2, sum, count)
+ (sum2, sum, count)
= (sum2, sum, count)
```

Which internally becomes:

```
  (sum2_0,          sum_0,         count_0          )
+ (sum2_1,          sum_1,         count_1          )
= (sum2_0 + sum2_1, sum_0 + sum_1, count_0 + count_1)
```

as there's a formula to get the standard deviation out of that:

> "the standard deviation is equal to the square root of the difference between the average of the squares of the values and the square of the average value."

```
STD = square root[1/N.Sum(x^2) - (1/N.Sum(x))^2]
```

Monoids don't have to use addition, here's an example of a monoid of ratios with the operation of multiplication:

```
public class Ratio
{
  private final int numerator;
  private final int denumerator;
  public static final Ratio NEUTRAL = new Ratio(1, 1);

  public Ratio(int numerator, int denumerator)
  {
    this.numerator = numerator;
    this.denumerator = denumerator;
  }

  public double ratio()
  {
    return numerator / denumerator;
  }

  public Ratio multiply(Ratio other)
  {
    return new Ratio(numerator * other.numerator, denumerator * other.denumerator);
  }
```

```
  ...// hashcode, equals, toString

}
```

Over the years I've grown the confidence that anything can be made into a monoid, with these kinds of tricks. Histograms with fixed buckets naturally combine, bucket by bucket:

The corresponding code for the add operation adds the number of elements in each respective bucket:

```
public Histogram add(Histogram other)
{
  if (buckets.length != other.buckets.length) {
    throw new IllegalArgumentException( "Histograms must have same size");
  }

  int[] bins = new int[buckets.length];

  for (int i = 0; i < bins.length; i++){
    bins[i] = buckets[i] + other.buckets[i];
  }

  return new Histogram(bins);
}
```

If histograms have heterogeneous buckets, they can be made to compose using approximations (eg curves like splines) that compose.

Moving average don't compose unless you keep all their respective memories and combine them just like the histograms. But by looking into small-memory microcontroller litterature you can find alternative ways to calculate them that compose with much less memory footprint, e.g. using just a couple of registers.

Note that one potential impediment to making an arbitrary calculation into a monoid could be concerns such as being ill-conditioned, or value overflow, but I never had this issue myself.

Monoids And Friends: Applications Notes

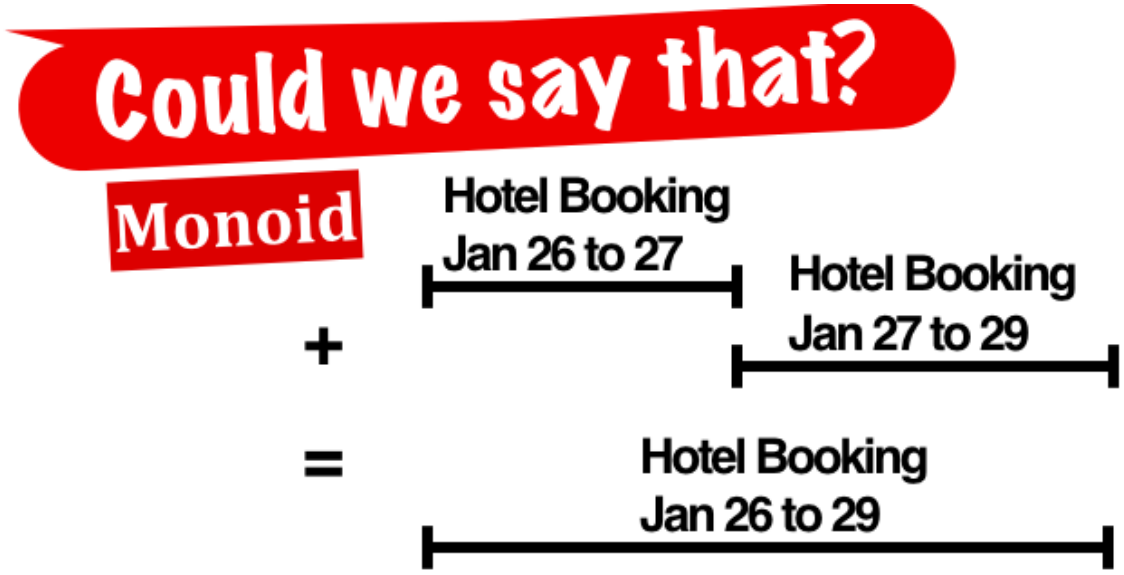
As shown with the pipes, monoids are ubiquitous in our daily lives, and are part of our universal language to describe things, even without ignoring their abstract definition. Everybody knows how to stack glasses or chairs. My kids know how to combine wooden trains together to create longer trains.

Declarative style

This ability to compose stuff is part of our mental models, and as such can be part of our Ubiquitous Language in the DDD sense. For example in the hotel booking domain, we could say that a booking from January 21 to 23 combined to another booking in the same hotel from January 23 to 24 is equivalent to one single booking from January 21 to 24:

```
  Booking [21, 21]
+ Booking [23, 24]
= Booking [21, 24]
```

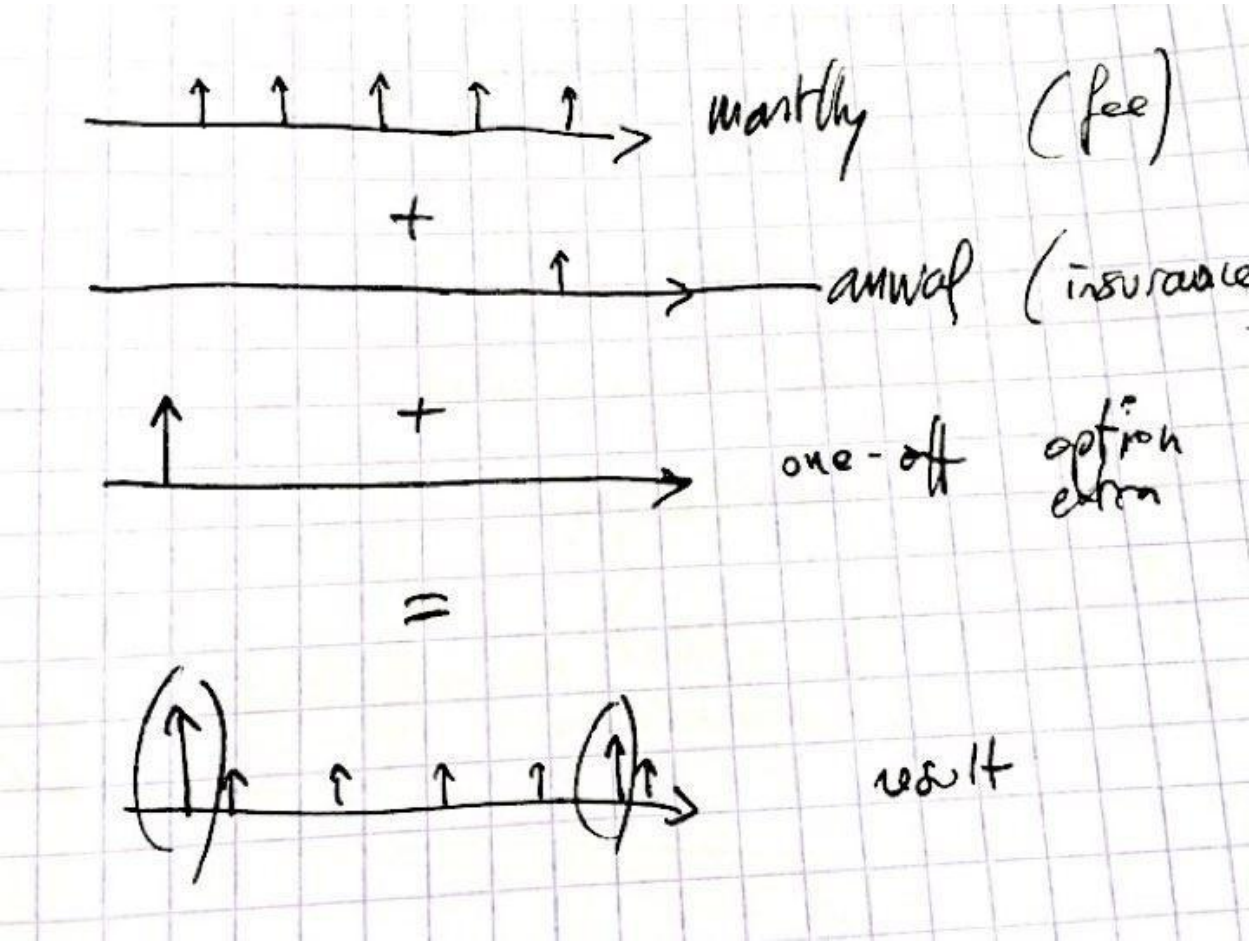
Which we could sketch like this, as ranges with some union operation:



The code for the operation would just take the min and max of both dates and check they share one date. It would probably be a commutative operation in this case.

Using monoids helps having a more declarative style in our code, another point that is advocated for by Eric Evans in the DDD book.

Let's consider another example of price plans of mobile phones. There's a potential fixed monthly fee, a potential annual fee for things like insurance, some potential one-off fees for extra options that you pay when you activate it etc. For a given price plan for a given customer, you have to select the cash flows sequences that match, then add them to create the invoice. We could draw the domain problem like this:



Unfortunately then developers tend to implement this kind of problem with an accumulation of special cases:

```
// without monoids
PaymentsFees(...)
PaymentsFeesWithOptions(...)
PaymentsFeesWithInsuranceAndOptions(...)
PaymentsFeesWithInsurance(...)
NoFeesButInsurance(...)
...
```

Whereas once you recognize that the cash flow sequences form a monoid, then you can just implement exactly the way you think about it:

```
// basic generators
monthlyFee(...) : Payments
options(...) : Payments
insurance(...) : Payments

// your custom code to combine

Payments invoice = monthlyFee
    .add(options)
    .add(insurance);
```

One major benefit is that the cognitive load is minimal. You just have to learn the type and its combine method, and that's it. And yet it gives you an infinite number of possibilities to combine them into exactly what you want.

Domain-Specific, within one Bounded Context

You may be tempted to reuse monoidal value objects across various parts of a larger system, but I would not advocate that. Even something as simple a Money class can be specific to some sub-domain.

For example in pretrade you would have a Money optimized for speed and expressed as an integer, as a multiple of the trading lot size, whereas for accounting you'd use a BigDecimal-based implementation that would ensure the expected accuracy even after summing many amounts and even after many foreign exchange conversions.

Another example this time with cashflows: in a tax-related domain, you can't just add an reimbursement cashflow to an interest cashflow, as they are treated very differently by the tax institution, whereas in an investment domain you would just add them all together without any constraint. For more on that point, I suggest Mathias Verraes' blog post[27] where he notes:

> *...dealing with money is too critical to be regarded as a Generic Subdomain. Different projects have different needs and expectations of how money will be handled. If money matters, you need to build a model that fits your specific problem space...*

Monoid, multiple times.

It's not uncommon for some domain concept to be a monoid more than once, for example once with addition and the neutral element ZERO, and a second time with multiplication and the neutral element ONE. As long as it makes enough sense from a domain perspective, then it's desirable to have more structures (being a monoid multiple times) or to be a stronger structure (see other mathematical structures later in this document).

[27] http://verraes.net/2016/02/type-safety-and-money/

Internal implementation hackery

Also note that neutral elements may call for some internal magical hacks for their implementation. You may rely on a magic value like `-1` or `Integer.MIN-VALUE`, or on some special combination of magic values. You may think it's bad code, and it would be if it was meant to be seen or used regularly. However as long as it's well-tested (or built from the tests) and as long as it's totally invisible for the caller of the class, then it will only cause harm when you are changing this class itself, which is probably acceptable. A monoid typically is not subject to a lot of changes, it's a finely tuned and highly consistent system that just works perfectly, thanks to its mathematical ground.

Established Formalisms, for Living Documentation

Monoids are one of the most frequent algebraic structures we can observe in the world of business domains. But other structures like groups (monoids with inverse elements), space vectors (addition with multiplication by a real number coefficient) and cyclic groups (think modulo) are also common. You can learn more about all these structures on Wikipedia and see whether they apply for your practical domain problems.

But the fact that these structures are totally described in the maths literature is important. It means that these solutions are established formalisms, which successfully passed the test of time. The DDD book actually advocates drawing on Established Formalisms for this reason.

Another reason is that you don't have to document them yourself. Just refer to the reference with a link and you're done. That's very much Living Documentation!

So if we want to document the fact that we implement a monoid, we could create a specific Java annotation `@Monoid(String neutralElement)`, that could then be used to annotate a combine method on some class:

Alternatively since Java 8 you could define a class-level annotation

```
@Monoid(neutralElement="emptyList", operation="union")
```

Since a class can be a monoid several times, you would also need to mark the custom annotation as `@Repeatable` and define its container annotation Monoids, so that you can then annotate a class multiple times:

```
@Monoid(neutralElement="one", operation="multiply")
@Monoid(neutralElement="zero", operation="add")
```

Self-Explaining Values

Now suppose you want a complete audit on all the calculations, from the initial inputs to the result. Without monoids you'll have a bad time going through all the calculations to insert logs at each step, while making the code unreadable and with plenty of side-effects.

But if the calculations are done on a type like a monoid, with custom operations, then you could just enrich the operations with internal traceability audit trail:

```
public static class Ratio
{
    private final int numerator;
    private final int denumerator;
    private final String trace;

    public Ratio multiply(Ratio other)
    {
        return new Ratio(
            numerator * other.numerator,
            denumerator * other.denumerator,
            "(" + asString() + ")*(" + other.asString() + ")"
        );
    }

    public String asString()
    {
        return numerator + "/" + denumerator;
    }
    // ...
}
```

With this built-in traceability, we can ask for the explanation of the calculation afterwards:

```
new Ratio(1, 3)
   .multiply(new Ratio(5, 2))
   .trace()
// trace: "(1/3)*(5/2)“
```

One question with the trace is to decide whether or not to use for the object equality. For example, is `(5/6, "")` really equal to `(“□, “(1/3)*(5/2)”)`? One way is to ignore the trace in the main object `equals()`, and to create another `strictEquals()` if necessary that uses it.

Encapsulated Error Handling

Many calculations can fail. We cannot divide a number by the number zero. We cannot subtract 7 from 5 in the set of natural integers. In software, error handling is an important source of accidental complexity, like in defensive programming, with checks statements bloating every other line of code.

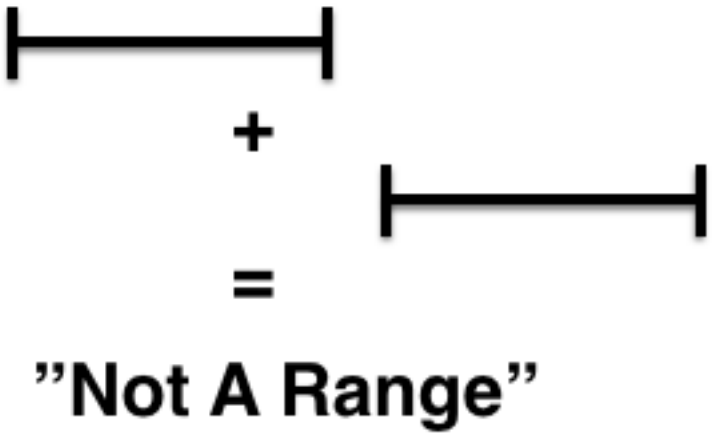
The traditional way to go to simplify is to throw exceptions, however it defeats the purpose or composability since it breaks the control flow. In practice I observe that throwing is acceptable for errors that are nothing but coding errors; once fixed they should never happen anymore, so for practical matters we have apparent composability.

An alternative to exceptions that can really happen at runtime is to make the monoidal operation a *total function*. A total function is a function that accepts any possible value for all its parameters, and therefore always returns a result for them. In practice the trick is to introduce a special value that represents the error case. For example in the case if division by zero, Java has introduced the special value `NaN`, for Not-a-Number.

Because a monoid has to follow the Closure of operation, it follows that the special extra value has to be part of the set of legal values for the monoid, not just as output but also as input. Usually the implementation of the operation would just bypass the actual operation and immediately return `NaN` when you get a `NaN` as a parameter: you propagate the failure, but in a composable fashion.

This idea was proposed by Ward Cunningham as the Whole Object pattern from his CHECKS patterns. Java Optional, and monads in functional programming languages, like the `Maybe` monad and its two values `Some` or `None`, are similar mechanisms to achieve this invisible propagation of failure. This is a property of an **Absorbing Element**[28] like `NaN: a + NaN = a`

In the case of ranges with the union operation, you may want to introduce a special element `NotARange` to represent the error case of the union of disjoint ranges:



In the case of natural integers and subtraction, the way to make the function total is to extend the set with negative integers (which at the same time will promote the monoid into a group with inverses) The way to make the square root function (nothing to do with a monoid) a total function would be to extend the set from real numbers to the superset of complex numbers.

How to turn anything into a monoid

This idea of extending the initial set with additional special values is a common trick for monoids, and not just for error handling. It’s also useful to artificially turn any set into one that is closed under a given operation.

[28] https://en.wikipedia.org/wiki/Absorbing_element

Given a function with some input `I` and output `O` that are not of the same type, we can always turn it into a monoid by introducing the artificial type that is the tuple of both types: `(I, O)`.

For example, given a function that gets a `String` and returns an `integer`, we can introduce the type `Something (String, int)`, so that we now have a function that gets a `Something` and also returns a `Something`.

Testing Monoids

As mentioned already, monoids are easy to test as they're immutable and have no side-effect. Given the same inputs, the *combine* operation will always return the same result. And with just one single function, the testing surface of a Monoid is minimal.

Still monoids should be tested on all their important properties, like being associative, and on some random values, with an emphasis on the neutral elements, any artificial value like `NaN` or Special Cases, and when approaching the limits of the set (`MAX_VALUE`…).

Since monoids are all about properties ("expressions that hold true") like the following:

```
Associativity
FOR ANY 3 values X, Y and Z,
THEN (X + Y) + Z == X + (Y + Z)

Neutral element
FOR ANY value X
THEN X + NEUTRAL = X
AND NEUTRAL + X = X

Absorbing element
FOR ANY value X
THEN X + NaN = NaN
```

Property-based Testing is therefore a perfect fit for testing monoids, since PBT tools can directly express and test these properties. For example in Java we can use JunitQuickCheck[29] to turn test cases into properties. Let us express the above properties for some custom Balance class with its neutral element and some absorbing element[30] called Error:

[29] https://github.com/pholser/junit-quickcheck
[30] https://en.wikipedia.org/wiki/Absorbing_element

```
public class Balance {
    private final int balance;
    private final boolean error;

    public final static Balance ZERO = new Balance(0);

    public final static Balance ERROR = new Balance(0, true);

    public Balance add(Balance other)
    {
        return error ? ERROR :
        other.error ? ERROR :
        new Balance(balance + other.balance);
    }
```

The properties could be written:

```
@RunWith(JUnitQuickcheck.class)
public class MonoidTest
{
    @Property
    public void neutralElement(@From(Ctor.class) Balance a)
    {
        assertEquals(a.add(ZERO), a);
        assertEquals(ZERO.add(a), a);
    }

    @Property
    public void associativity(
            @From(Ctor.class) Balance a,
            @From(Ctor.class) Balance b,
            @From(Ctor.class) Balance c)
    {
        assertEquals(a.add(b).add(c), a.add(b.add(c)));
    }

    @Property
    public void errorIsAbsorbingElement(@From(Ctor.class) Balance a)
    {
        assertEquals(a.add(ERROR), ERROR);
        assertEquals(ERROR.add(a), ERROR);
    }
```

The PBT tool will run these test cases for a number (the default being 100) of random values.

Beyond monoids

When we model real-life domains into software, we most frequently recognize Monoids as the underlying mathematical structures that best matches the domain as we think about it.

But there are many other mathematical structures that are valuable to know. To be fair, you don't have to know their esoteric names to use them, you just have to focus on their respective distinguishing feature, or I should say "distinctive property".

Here are some common algebraic structures, each with their name (useful if you want to impress people) and most importantly the specific property that makes them special.

If you just have the closure of the operation, then it's called a "**magma**", don't ask me why. It's a much weaker structure than monoids. If you also have associativity, then it's called a "**semigroup**". If you add the neutral element, then it becomes a **Monoid** indeed. And from that we can keep on adding specific properties. Note that all these structures are all about composition, plus something that helps composition even more.

If for any value there exists an *inverse value*, then the monoid becomes a "**group**":

```
value + inverse-value = neutral element.
```

It's a strong property to have inverses for all values. For example, natural integers don't have inverse with respect to addition, but signed integers do: `3 + (-3) == 0`. In business domains, having inverses is less frequent, so groups are less frequently used than monoids. Groups are all about compensation, with inverse values that can always compensate the effect of any value.

Going further, if we can compose not just whole elements but also compose values *partially*, then we have a **vector space**. For example we would write that `(1, 3) + 0.5(6, 8) = (4, 7)`. Notice the coefficient (the real number 0.5 here) that modulates the impact of the second term. Money with addition can be seen not just as a monoid, but as a group, and even as a space vector:

```
  EUR25
+ 1.5.EUR30
= EUR70
```

Space vector is all about addition with multiplication by a scalar coefficient.

Any structure that happens to yield the same result regardless of the ordering of the values is called "**commutative**": `a + b = b + a`. This is a very strong property, not so frequent, so don't expect it too much in domain modeling, but if you see it or manage to make it so, go for it. Commutativity helps a lot especially for situations of "out of order" events, for example when distributed systems communicate through a network, it's frequent for events a, b, c that used to be in this ordering to

arrive out of order, e.g. b, a, c. Being commutative is the most elegant way to deal with that. Exotic structures like CRDT[31] rely on commutativity for that, but it's far beyond the scope of this text.

There's another structure that I like a lot. It's a very simple yet common one, and is called the "**Cyclic Group**", and its key idea is the modulo, hence the name cyclic. The days of the week form such a cyclic group of order 7 (its size), and the months of the year are another of order 12. Cyclic groups have a finite number of values, and when you reach the last value your cycle back to the first one: if the order of the cyclic group is 3, then the values are `{0, 1, 2}` and `2 + 1 = 0`.

Numeration and time love cyclic groups, and as a result, domain-specific numeration and time also love them. For example, in finance, financial derivatives like options and futures are identified by their expiry date, simplified as a month code and a year code, e.g. H9 means March 2019, but also March 2029 or March 2009. The letter codifies the month, and the number codifies the year. They're both cyclic groups (one of order 12, and the other of order 10). And it turns out that the product of both is also a cyclic group of order 12*10 = 120, that's what we can learn by looking Wikipedia on Cyclic Groups[32]. One benefit from using established formalism is that it comes with a lot of proven properties and theorems we can rely on safely. One of interest is that every cyclic group of order N is isomorphic (think equivalent) to the one on integers of order N, called Z/nZ. This means in practice that you can always implement it with integers as the internal state and modulo arithmetics.

Then there are many other more complicated algebraic structures available that deal with more than one operation and how the operations interact: a **ring** for example generalizes the arithmetic operations of addition and multiplication. It extends a commutative group (addition) with a second operation (multiplication), and requires that this second operation distributes with the first one:

```
a.(b+c) = a.b + a.c
```

Over the past 15 years I've created my own domain-specific values from all the above-mentioned structures, and many times without knowing the corresponding name. Still, it helps to pay attention to the properties that we can build upon or not, for a recap, with a given operation noted "+":

- **closure of operation** `T + T` is a `T`
- **associativity**: `a + (b + c) = (a + b) + c`
- **neutral element** `e` such that `a + e = a`
- **inverse** `(-a)` for any `a` such that `a + (-a) = 0` (your own zero)
- **using a coefficient**: `a + alpha.b`
- **commutativity** : `a + b = b + a`
- **cycle** of order `N`: `a + N = a`

[31] https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type
[32] https://en.wikipedia.org/wiki/Cyclic_group

Inheriting the algebraic properties from the implementation structures

We usually implement domain-specific concepts from the standard built-in types of the programming language: boolean, integers and other numbers, finite sets of enums, Strings, and all kinds of lists. It happens that all these types exhibit many properties: numbers are rings, groups, space vectors, enums can be seen as Cyclic Groups, boolean are groups, lists and maps can easily be seen as monoids. And it happens that putting several structures next to each other as fields (product types) usually preserves the relation if the operation on the whole is defined as the field-wise operation of each of the components. This explains why so many domain concepts *inherit* part of their internal implementation structure, unless you mess with their operation. Think about it when you implement.

Make your own arithmetic to encapsulate additional concerns

Creating your own arithmetic helps keep your code simple even when you need to perform calculation of a value "with something else", like keeping track of the accuracy of the calculation, or of its uncertainty, or anything else. The idea is to expand the value into a tuple with the other thing you also are about:

```
(Value, Accuracy)
(Value, Uncertainty)
```

And to expand the operation into the tuple-level operation, trying to preserve some desirable properties along the way.

For example for the type `TrustedNumber(Value, Uncertainty)` you could define the addition operation this way, in a pessimist fashion such as the resulting uncertainty is the worst of both operands:

```
public add(TrustedNumber o)
{
  return new TrustedNumber(
      value + o.value,
      max(uncertainty, uncertainty)
  );
}
```

This approach is standard in mathematics, for example for a complex numbers, or dual numbers.

Creating your own arithmetic is more natural and more good-looking with operator overloading, which does not exist in Java.

For more examples on how drawing on established formalisms and algebraic structures, don't hesitate to dig into JScience[33]; I did a decade ago and I learnt a lot from it. It's built on a *linear*

[33] http://jscience.org/api/org/jscience/physics/amount/Amount.html

algebra layer of supertypes, from which everything else is built upon.

Case Study: Environmental Impact Across a Supply Chain

Just like other code snippet across this article, the code for this case study is online[34].

Putting together all what we've seen so far, we will study the case of a social network to track the environmental impact of companies and their suppliers.

Let's consider a pizza restaurant willing to track its environmental impact across its complete supply chain. Its supply chain can be huge, with many direct suppliers, each of them having in turn many suppliers, and so forth. The idea is that each company in the supply chain will get invited to provide its own metrics, at its own level, along with the names of its direct suppliers. This happens massively in parallel, all around the world, across potentially hundreds of companies. And it also happens incrementally, with each supplier deciding to share their impact when they become able or willing to. Still, at any time, we would like to compute the most up-to-date aggregated impact for the pizza restaurant at the top.

The impacts we are interested in include the **number of suppliers** involved for one pizza, the **total energy consumption** and **carbon emission** by pizza produced, along with the respective **margins of error** for these numbers, and also the **proportion of certified numbers** (weighted by their respective mass in the final product) over the whole chain.

We could collect all the basic facts, and then regularly run queries to calculate the aggregated metrics over the whole dataset each time, a brutal approach that would require lots of CPU and I/O. Or we could try to start from what we already had and then extending it with the latest contributions in order to update the result. This later approach can save a lot of processing (by reusing past calculations, in addition to enabling a map-reduce-ish approach), but requires each impact to compose smoothly with any other.

We decide to go the later route. We want to compose, or "chain" the impacts together all across the chain, to compute the full impact for one pizza in our restaurant at the root of the supply chain.

From what we've seen, we need to define a concept of Environmental Impact that:

- can represent the metrics available for one supplier in isolation
- can represent the metrics that matter for the pizza restaurant in terms of impact at the top of the supply chain
- can compose all supplier's metrics, and their supplier's, into the aggregated metrics for the pizza restaurant.

Out of the impacts we want, the number of suppliers is easy to compose: for each supplier (level N), its supplier count is exactly 1 plus the supplier counts of all its direct suppliers (level N-1). It's

[34] https://gist.github.com/cyriux/a263efb9c483bcefe72e49c3343ff24e

naturally additive, in the simplest possible way. The energy consumption and carbon emissions are naturally additive too. This suggests the following concept in code:

```
public static class EnvironmentalImpact
{
  private final int supplierCount;
  private final Amount energyConsumption;
  private final Amount carbonEmission;

  // … equals, hashcode, toString

}
```

Now in order to compose partial impacts in a way that is weighted by their respective contribution to the pizza, we make this value a space vector, with the "addition" and "multiplication by a scalar" operations:

```
public EnvironmentalImpact add (EnvironmentalImpact other)
{
  return new EnvironmentalImpact
      supplierCount + other.supplierCount,
      energyConsumption.add(other.energyConsumption),
      carbonEmission.add(other.carbonEmission)
  );
}

public EnvironmentalImpact times (double coefficient)
{
   return new EnvironmentalImpact(
      supplierCount,
      energyConsumption.times(coefficient),
      carbonEmission.times(coefficient)
   );

}
```

Because all these amounts are not that easy to measure, they come with significant margins of error, which we'd like to track when it comes to the end result. This is specially important when suppliers don't provide their impact, so we have to guess it, with some larger margin of error. This could make the calculations quite complicated, but we know how to do that in a simple way, using another tuple that gathers the amount, its unit and its margin of error:

```
public static class Amount
{
  private final double value;
  private final String unit;
  private final double errorMargin;
  // … equals, hashcode, toString
}
```

And because we want to add these amounts weighted by coefficients, we want to make it a space vector as well, with the addition and multiplication by a scalar:

```
public Amount add(Amount other)
{
  if (!unit.equals(other.unit))
      throw new IllegalArgumentException(
        "Amounts must have same units: " + unit + " <> " + other.unit
      );

  return new Amount(
      value + other.value,
      unit,
      errorMargin + other.errorMargin
  );
}

public Amount times(double coefficient)
{
    return new Amount(
        coefficient * value,
        unit,
        coefficient * errorMargin
    );
}
```

We're lucky the error margins are additive too. But it's also possible to calculate them for any other operation than just addition if we wanted to.

Now we're almost done, but remember we wanted to track the proportion of certified numbers in the whole chain. A proportion is typically expressed in percentage, and it's a ratio. If we compose one impact that is 100% certified with two other that are not at all, then we should end up with a proportion of certification of 1/3, i.e. 33%. But we want this proportion to be weighted by the respective mass of each supplier in the final product. We notice that this kind of weighted ratio is not additive at all, so we need to use the trick of making it into a tuple: (total certification percents, total of the weights in kg), which we can compose with addition and multiplication by a scalar.

So we now decorate the Amount class with a `CertifiedAmount` class that expands it with this tuple:

```
/** An amount that keeps track of its percentage of certification */
public static class CertifiedAmount
{
  private final Amount amount;

  // the total certification score
  private final double score;

  // the total weight of the certified thing
  private final double weight;
```

And we update our `EnvironmentalImpact` class to use the `CertifiedAmount` instead of the Amount, which is easy since it has the exact same methods names and signatures.

Now let's use that for 1 pizza, that is made of 1 dough, 0.3 (kg) of tomato sauce and some cooking in the restaurant.

```
EnvironmentalImpact cooking = singleSupplier(
  certified(1, "kWh", 0.3), // energy
  certified(1, "T", 0.25)   // carbon
);
EnvironmentalImpact dough = singleSupplier(
  uncertified(5, "kWh", 5.),
  uncertified(0.5, "T", 1.)
);

EnvironmentalImpact tomatoSauce = singleSupplier(
  uncertified(3, "kWh", 1.),
  certified(0.2, "T", 0.1)
);
```

Which displayed into the console:

```
EnvironmentalImpact(
  1 supplier,
  energy: 1.0+/-0.3 kWh (100% certified),
  carbon: 1.0+/-0.25 T (100% certified)
)

EnvironmentalImpact(
  1 supplier,
  energy: 5.0+/-5.0 kWh (0% certified),
  carbon: 0.5+/-1.0 T (0% certified)
)

EnvironmentalImpact(
  1 supplier,
  energy: 3.0+/-1.0 kWh (0% certified),
  carbon: 0.2+/-0.1 T (100% certified)
)
```

From that we can calculate the full impact of the restaurant by chaining each impact:

```
EnvironmentalImpact pizza = cooking
  .add(dough)
  .add(tomatoSauce.times(0.3)
);
```

If we print the resulting impact into the console, we get:

```
EnvironmentalImpact(
  3 suppliers,
  energy: 6.9+/-5.6 kWh (43% certified),
  carbon: 1.56+/-1.28 T (56% certified)
)
```

Which is what we wanted. We can then extend that approach for many other dimensions of environmental impact accounting, more details on accuracy, estimated vs measured vs calculated values, traceability of the numbers etc., just by expanding the concepts at each level, while still keeping it all as nested mathematical structures that compose perfectly. This approach scales for high complexity, and for high cardinality as well.

Domain-Driven Design loves monoids

Domain-Driven Design leans towards a functional programming style in various aspects. The most visible is the obvious Value Object tactical pattern, but in the Blue Book you can also find the patterns

Side-Effect-Free Functions, Closure of Operations, Declarative Design and Drawing on Established Formalisms.

It turns out that if you put all of them together, you end up with something like monoids.

Monoids are everywhere, even in Machine Learning, with the ubiquitous matrices and tensors, and with the key trick of composing derivatives together thanks to the Chain Rule[35].

Once you've used monoids a few times you can't but fall in love with them. As a consequence, you try to make everything into a monoid. For example with my friend Jeremie Chassaing we've discussed monoids and Event Sourcing, and he kept investigating how to make Monoidal Event Sourcing[36].

The code for the code snippets in this text are all online as Github gists[37].

Many thanks to my colleague Mathieu Eveillard for reviewing an early draft, and to reviewers Eric Evans and Mathias Verraes

[35] https://en.wikipedia.org/wiki/Chain_rule
[36] https://thinkbeforecoding.com/post/2014/04/11/Monoidal-Event-Sourcing
[37] https://gist.github.com/cyriux

Enhancing DDD — Prof. David West

In celebration of the fifteenth anniversary of Eric Evans' Domain-Driven Design

I would like to communicate two things in this essay: first, the profound contribution that Eric Evans has made to software development; and second, pose some questions that might lead to explorations and possible enhancements. Before doing either of those, it is useful to provide some context.

A Bit of History

Computers were new then, and inspiring. Vannevar Bush, *As We May Think*, imagined a "Memex," a hypertext type device to augment research and thought. Douglas C. Engelbart wrote of *Augmenting Human Intelligence*. Alan Kay envisioned the *Dynabook*. Steve Jobs (not a contemporary) had a vision of a *bicycle for the mind*.

Computers are ubiquitous today, and perplexing. What happened to all that potential? How did visionary utopias morph into Facebook as the exemplar of the advertising dominated dystopia described by Frederic Pohl and C.M. Kornbluth in *The Space Merchants* (circa 1955)?

Software was new then, and valuable. COBOL programs of 1-5,000 lines that took a few weeks to conceive, implement, and deploy reduced costs, or increased profits, by hundreds of thousands of dollars. Slightly larger software systems created strategic advantage worth millions. Work was enhanced by software that automated its most tedious and mundane aspects, allowing the worker to utilize her innate human abilities to a greater extent.

Software is pervasive today, and costly. Massive, convoluted, and complicated software inhibits business agility and innovation. Trillions of dollars are spent annually, ninety-percent of it on maintenance of legacy code, on software that is, at best, a commodity. Work is increasingly demeaning to the humans doing it as they are reduced to mere 'machine parts' with no freedom to think or act independently of how the computer instructs them.

Addressing all the whys and wherefores of how dystopian futures replaced utopian is beyond the scope of this essay, but how software and software development devolved is more tractable.

Let us begin with the LEO I — the world's first business computer.

LEO, (Lyons Electronic Office), was built by J. Lyons and Company, a leading catering and food manufacturing company in the UK, in 1951. The same team built the hardware, programmed system software, and programmed a set of applications that included: payroll, order entry, inventory control, production scheduling, and management reports. The computer was even used to create customized tea blends — a rudimentary "expert system."

A number of unexamined assumptions were made while constructing LEO that contributed significantly to the problems with software development that bedevil us today. Among them:

- Black Box programming. Programming was defined exclusively in terms of the computer, the machine. A precisely defined set of inputs entered the black box and a precisely defined set of outputs emerged. The programmer had to envision all the states and state transitions — within the box — that assured the correct correlation of inputs and outputs. Several decades later, Fred Brooks, in his "No Silver Bullet" paper, identified the challenge of mentally keeping track of all these states and transitions as THE essential problem of software development.
- Integrated, monolithic, systems. This one would be hard to avoid, given that there was only one computer available.
- Centralized hierarchical control. Epitomized by the ubiquitous 'Program Structure Chart' with a single master control module in charge of subordinate afferent, transform, and efferent modules.

The most egregious, albeit understandable, assumption concerned the domain; which was assumed to be no different, in essence, than the computer itself. That is to say that the domain — clerical tasks — was assumed to be a deterministic mechanical system of the same type as the computer itself. When the domain is purely clerical tasks, this assumption is not unreasonable. For the next 15-20 years, the era of "Data Processing," few software development efforts challenged the assumption.

Little, if any, thought was given to a domain, some subset of the natural world (e.g. a business or business process). It was not until the early seventies when the era of Data Processing gave way to the era of Management Information Systems that any attention was paid to the domain.

In the early 1970s the ideas and practices of "Structured Analysis and Design" (SAD) dominated the practice of software development. SAD advocated:

- Step One: model the domain
- Step Two: determine what changes you wanted to enhance or correct issues in the domain.
- Step Three: brainstorm multiple ways in which the changes might be affected.
- Step Four: analyze which 'solution' was optimal.
- Step Five: model the chosen solution — including modularization of the software.
- Step Six: implement the model.
- Step Seven: deploy and evaluate the implemented solution and its impact on the domain

Seldom, if ever, were Steps One through Four actually performed. In part, because business management failed to see the value and therefore discouraged or forbid them as a "waste of time."

The most important factor in the demise of the domain was "Software Engineering." In 1968 a new profession and new academic discipline were defined. Software engineering was to be applied Computer Science in the same way that Structural engineering was applied math and physics.

To be a professional software engineer you needed to know everything possible about the computer. Programming was nothing more than the applied science of "algorithms plus data structures." (Dykstra's famous definition of programming.)

A professional software engineer began work with a set of "complete and unambiguous" requirements and then built a piece of software that provably satisfied those requirements. All of the

tools and techniques utilized by the software engineer were focused facilitating the programmer's understanding of what was going on inside of the computer and/or structuring and inter-relating software modules within the context of the computer.

By the mid 1970s, the Domain was Dead.

Life Underground

Only to the Mainstream was the Domain lost.

In 1985, Peter Naur tried to tell us why and how the domain was critical to software development. Railing against the prevailing " software engineering production model" of software development, Naur insisted that developers were engaged in a process of theory building. A theory of, "***an affair of the world*** *and how the software will handle and support it.*" [emphasis mine]

Lost in the rush towards Object-Oriented Programming in the 1980s and 1990s were critical aspects of the Object Idea. In OOP, objects were nothing more than animated data structures, a means of modularizing a programs source code. In contrast the Object Idea was to use the criteria of "responsibilities" to modularize and understand the Domain and to create a common vocabulary, a common ontology, that would map domain modules to software modules.

User stories, ala Kent Beck and Extreme Programming, were yet another attempt to bring the Domain back into the development process. "On-site Customer" and the practice of "Exploratory, Iterative, Development" brought the Domain front and center while simultaneously providing practices and principles that supported Peter Naur's vision of theory building.

Challenges presented by "Ultra-Large Scale Systems" and "Complex Adaptive Systems" have taken center stage in the past decade. As our understanding of these systems increases, so to is the conviction that natural, Domain, systems are qualitatively different from the simple deterministic system of computer plus software. The assumption made with the building of LEO — that the domain was just as deterministic and simple as the computer — is false.

These, and numerous other, advocates for the domain were, seemingly, ignored by mainstream developers. Practitioners made the attempt, but barriers like preventing programmers from talking to users — only business analysts could do that, and they could only talk to system architects — kept the domain at bay.

And, of course, for software engineering the domain was irrelevant. Users provided specifications / requirements and developers coded to satisfy requirements. Job done.

Comes the Dawn

My first impression, opening Eric Evans' *Domain-Driven Design*, was pleased amazement that someone was actually paying attention and was aware of the underground existence of the Domain.

Martin Fowler's Introduction and Eric's own words assert that a robust domain model is essential for the success of any software project. They even asserted that there was an entire community within software development that understood the essential value of domain knowledge and domain models.

Aaah, but how to conceive and construct such models?

Eric proceeds to provide us with principles and practices, techniques and tools, to do just that. Readers of this essay will already be familiar with the contents of the original book and subsequent developments of DDD, so it is not necessary to attempt a recap of specifics here.

The amazing thing about DDD was not the patterns or the practices, it was the quiet way it put the lie to a fundamental tenets of software engineering: the lie that programmers did not need to have an understanding of domains, that everything they needed to know was a set of requirements that the code must satisfy.

At the time DDD was written the prevailing trend in software development organizations, especially large one, was establishment and enforcement of communication silos. Business Analysts were the only ones allowed to communicate with users or domain experts — Architects were the only ones allowed to communicate with Business Analysts — Systems Analysts and Data Base Administrators were the only ones allowed to communicate with Architects —Designers were the only ones allowed to communicate with System Analysts / Data Base Administrators — and Testers were the only ones allowed to communicate with Designers.

Coders were limited to receiving one-way communications from Designers, in the form of specifications, and Testers, in the form of bug reports.

By 2004 it was becoming clear that an integral and essential aspect of Agile development — stories written by domain experts / users in their own vernacular and On-Site Customer — were not going to be allowed by management. Indirect communication filtered through a "product owner" isolated the development team from domain experts and 'users'.

Not only did Eric challenge prevailing wisdom regarding the domain, he demonstrated precisely why an understanding of the domain was essential to successful implementation.

A tertiary contribution, was filling in some of the 'design gaps' of iterative, incremental, (Agile) approaches to development. For example: XP provided clear directives for building and using domain models but relied on the code (as noted by Eric) as the sole implementation model. Kent Beck and the early practitioners of XP were already expert designers and they had deep implicit knowledge of how to transition from a user story to code. But they never spelled out the nature of that "magic" in any way that would allow less adept developers to follow their example.

Eric provided some intellectual "middleware" that addressed this problem.

It would be difficult to understate the importance of DDD and Eric's contributions — both with the original book, papers written, and via his active involvement in discussion forums and conferences.

My own appreciation of his work had led me to ask questions; questions I would love to have the opportunity to discuss with Eric and the many experts in DDD that attend conferences like the

Domain-Driven Design – Europe conference in Amsterdam. It is my naïve belief, and hope, that thinking about this kind of question might lead in directions fruitful for enhancing DDD.

Modeling

Both Eric (in the first few pages of section I) and Martin Fowler (in his foreword) assert the need for both a Domain Model and an Implementation Model and for consistency between them. This is an important observation.

Just how important can be illustrated by three prior failures; one with objects and the second with Agile.

For responsibility-driven object projects the domain model was a set of CRC (Class, Responsibility, Collaborator) Cards. Great effort was expended in creating a set of cards that could be used to verbally walk-through a set of stories about object interactions in pursuit of some goal. Care was taken to assure that each card listed only those responsibilities appropriate for the object and that responsibilities were distributed across the set of objects in an optimal way.

But then what? How did you get from a 3x5 card with a list of domain natural responsibilities [e.g. "provide age upon request" on a Person card] to code [e.g. (SystemClock today – self DOB) asInteger]? In my own work, I invented and utilized an "Object Cube" that extended the CRC card to include, a Knowledge Required, a Protocol, and an Events section, all created from the domain perspective. The addition of these elements to the domain model allowed the programmer to directly write the code to implement the objects.

When Kent Beck (and Ward Cunningham) introduced Extreme Programming, the domain model was the user story, or, more accurately the collection of stories in the Product Backlog. Code was the implementation model. Never explained was how you got from a User Story to Code. And when management disallowed the practice of On-site Customer and, to a great extent, stories actually written by domain experts, the results were predictable.

In typical software engineering, the domain model was an extensive set of "requirements." Multiple implementation models (e.g. the set advocated by UML) were generated with tenuous, if any, connection to an actual model of the domain. Software was written to "satisfy requirements." The vast volume of outsourced and offshored software that demonstrably, even provably, 'satisfied requirements' while remaining unusable is testimony to the failure of this approach. [Not to pick on outsourcing or offshoring, but projects that remained in-house had many opportunities to cheat by going around impediments to seek fuller, more semantic, understanding of the domain than was ever available to those in remote locations.]

Eric's book explicitly and implicitly addresses this issue by having the developer / development team create the domain model in cooperation with and in conjunction with domain expert(s).

This approach brings to mind two interesting and inter-related questions.

First, is the domain model truly a 'domain model'? Or, rephrasing, would the domain experts, absent the influence and direction of the developer, have created the same, or substantially the same model?

I read a lot of business books. Most of them propose models to facilitate the understanding of the business domain. None of these models, excepting PERT charts, have any semblance to any of the standard UML models.

The related question concerns the utility of the domain model, constructed from the perspective of the developer, that Eric asserts is essential for the ongoing success of the development team in their work; is it equally valuable to the domain expert for increasing their understanding of the domain, how it works, and how it might be evolved? Or, as I believe to be the case, that when she is not working with a software team, the domain expert uses other, more familiar, models within the domain?

A corollary question: is it possible for any formal or even semi-formal model to capture the true complexity of any natural domain? (Exempt from this question are internal system domains like device drivers and purely clerical, number crunching, domains like the earliest examples of payroll and order entry, ala LEO in 1951.)

Natural domains, are, most often, complex, highly dynamic, ambiguous, and self-contradictory. An implementation system may be convoluted and complicated but it cannot be complex. Implementation systems are, necessarily, unambiguous, precise, and quantifiable. Is any semi-formal modeling process, grounded in the perspective and world view of the developer, adequate?

None of these questions are intended to fault the kind of models advocated in Eric's book nor should the be construed in any way to diminish the immense contribution he has made to software development.

Instead they should be seen as asking if we might enhance domain modeling on behalf of software development by exploring the use of story, visualization - like iconic Tibetan Mandalas, metaphor, and the simple system models of General Systems Theory.

Language

As noted in the previous section, DDD insists on domain models and implementation models with significant consistency between them. Is not source code an implementation model? Perhaps it is the ultimate implementation model. As Eric noted, in most Agile approaches, the source code is the only implementation model.

Eric also makes an important point concerning language and the assertion that the team should share ***one*** language. By this he means that the domain expert and the developer use the same language. It also argues that the multiple specialty argots of tester versus DBA; Java programmer versus JavaScript programmer; framework expert and tool maven; etc. be unified in some important way in a common language embedded in the domain and implementation models.

Should any 'common language' extend past any visual models to include the source code itself? The question is not raised in Eric's book and I do not know if it has come up in discussions, forums, blogs, etc. since the book was published.

My personal answer is yes!

Early in my career I wrote application programs in COBOL. (System level programs were written in Assembler.) We often had domain experts, bankers in that instance; sit in on 'code reviews'. We also had domain experts, bank auditors, read and review code before it went into production as a means of detecting or preventing fraud. COBOL made this easy.

Decades later, coding in Smalltalk, it was also possible for domain experts to read and comment on, sometimes enhance, source code. While it is true that domain experts were seldom exposed to large parts of the class library, and never the tiny-C implemented kernel code, they did see the code that implemented objects and the code (usually in Workspace) that exposed object interactions in pursuit of application goals.

I could cite numerous examples similar to one where a domain expert, participating mostly as an observer, responded to a "does not understand" error message, by waving a CRC card and telling the coders, "wait, we forgot to account for this collaboration before we sent that message. Without the collaboration we do not have an object in place to receive the message we are trying to send.

Smalltalk was expressly designed to be a shared language — between human and the computer — that would allow domain experts to express their needs without having to think about how the computer would, internally, satisfy those needs. Simula, before it became Simula I and a programming language, had the same goal; to allow the domain expert to "discuss" the 'what' and 'why' of a program without ever concerning themselves with the 'how'.

I am not suggesting any kind of "end user programming" by saying these things as there is a lot of 'how' knowledge required to successfully develop software. But, I am suggesting, strongly, that source code be part of the implementation model and that the common language shared between domain expert and developer should include much of the programming language.

While this is clearly possible with languages like COBOL and Smalltalk (and for more scientific domains, FORTRAN) I can categorically assert that it is not possible in languages like C, C++, LISP, or any functional language. It is improbable with Java.

The Heart of Software

> "The heart of software is its ability to solve domain-related problems for its user. All other features, vital though they may be, support this basic purpose."

Eric Evans, Domain-Driven Design, page 4

At the heart of this simple statement is another gauntlet thrown at the feet of traditional, mainstream, software engineering. Another example of Eric challenging prevailing wisdom.

In fifty years of practicing, observing, and teaching software development: I can count on two hands the number of times any approach, method, or theory of software development has asserted the critical importance of software solving domain problems.

Acceptance and Usability testing and User Experience (UX) design might be construed as indirect measures of the heart of software but not more. [That is a discussion for another time and place.]

As much as I agree with Eric's words and their underlying premise: questions arise.

Should our domain model extend to include the forces and constraints that gave rise to the problem in the first place? If it does not, how can we differentiate between A solution and An Optimal solution? Does it matter?

A corollary: without an understanding of the forces at work in the domain, will we have any way to anticipate both advantageous and disadvantageous reactions of the domain system to the introduction of this new software artifact? Again, does it matter?

As an anthropologist I have extensively studied "technology and cultural change." A culture is a complex system, so to is a business, so to are all the myriad domains in which and for which we develop software.

The introduction of any change in such a system forces the system out of equilibrium and it changes, in unpredictable ways, in order to establish a new state. Some examples from culture: the invention of clay pots singularly contributed to the movement from hunter-gatherer societies to villages; and, the introduction of the automobile led to massively distributed cities and the sexual revolution.

From the history discussed above, LEO totally transformed the world of business and, eventually, gave rise the massive data stores and associated concerns about privacy and targeted marketing.

Surely, we cannot, and do not want, our domain model to be extensive enough to account for the entirety of the natural system where we will introduce our software. However, my personal belief is that our domain models must incorporate some understanding of how our well-crafted solution affects some of the elements in the domain - specifically and emphatically, the People.

Our software may very well "solve a domain related problem for its user," but at what cost to the user?

Will our solution totally displace the user? I can remember a time when the development of new software was cost justified by the number of workers that would be fired once it was implemented.

Or will it simply make the user's work a living hell? Sometime you might ask a gate attendant trying to reschedule a 250 irate passengers of a cancelled flight, just how much the software she is mandated to use "helps" her and makes her job easier and more pleasant.

These questions might be distilled to a general one, "to what degree to ethics, morality, and respect for human beings mandate the extension of our domain model into areas not obviously connected to development of the software itself."

Challenge

Fifteen years ago, Eric pulled together a collection of inter-related software development issues and means for addressing those issues under an umbrella concept stressing the primacy of domain understanding and domain modeling.

This book and his work putting it together should be regarded as a major and important effort. It should also be seen as a challenge.

Can the community that has enthusiastically adopted DDD transcend simple implementation of the ideas and practices advocated in 2004 and work with him, and each other, to enhance and extend; to build on the foundation provided to us.

Are you building the right thing? — Alexey Zimarev

Introduction

This essay is the first chapter of the Hands-On Domain-Driven Design with .NET book

The software industry appeared back in the early 1960s and is growing ever since. We have heard predictions that someday all software would be written and we will not need more software developers, but this prophecy has never become a reality, and the growing army of software engineers is working hard to satisfy continually increasing demand.

However, from the very early days of the industry, the number of projects that were delivered very late and massively over budget, plus the number of failed projects is overwhelming. The 2015 CHAOS Report by Standish Group[38]) suggests that from 2011 to 2015 the percentage of successful IT projects remains unchanged at a level of just 22%. Over 19% of all projects failed, and the rest have experienced challenges. These numbers are astonishing. Over four decades a lot of methods have been developed and advertised to be a silver bullet for software project management, but there is no or little change in the number of successful projects.

One of the critical factors that define the success of any IT project is understanding the problem, which the system to be designed, suppose to solve. We all very familiar with systems that do not solve problems they claim to answer or do it very inefficiently. Understanding the problem is also one of the core principles of the Lean Startup methodology, proposed by Eric Ries in his book *The Lean Startup* by Crown Publishing. Both Scrum and XP software development methodologies embrace interacting with users and understanding their problems.

Domain-Driven Design (DDD) term was coined by Eric Evans in his now-iconic book *Domain-Driven Design, Tackling Complexity in the Heart of Software* by Addison-Wesley back in 2004. More than a decade after the book was published, interest in practices and principles, described in the book, started to grow exponentially. Many factors influence such growth in popularity, but most important one is that DDD explains how people from software industry can build an understanding of their users needs and create software systems, which solve the problem and make an impact.

Understanding the problem

We rarely write software to write some core. Of course, we can create a pet project for fun and to learn new technologies, but professionally, we build software to help other people to do their work

[38] https://www.projectsmart.co.uk/white-papers/chaos-report.pdf

better, faster, and more efficiently. Otherwise, there is no point in writing any software in the first place. It means that we need to have a *problem*, which we intend to solve. Cognitive psychology defines the issue as an obstacle between the current state and the desired state.

Problem space and solution space

In their book *Human Problem Solving* (1972, Englewood Cliffs, N.J.: Prentice-Hall), Allen Newell and Herbert Simon outlined the problem space theory. The theory states that humans solve problems by searching for a solution in a *problem space*. The problem space describes these initial and desired states and possible intermediate states. It can also contain specific constraints and rules that define a context of the problem. In the software industry, people operating in the problem space are usually customers and users.

Each real problem demands a solution, and as soon as we search good enough in the problem space, we can outline which steps are we going to take to move from the initial state to the desired state. Such an outline and all the details about the solution form a *solution space*.

The classical story of problem and solution spaces, which get completely detached from each other during the implementation, is the story of writing in space. The story says that in 1960s space nations realised that usual ball-pens wouldn't work in space due to lack of gravity. NASA then spent a million to develop a pen that would work in space, and Soviets decided to use good old pencil, which costs almost nothing.

This story is so trustworthy that it is still circulating and was even used in the "West Wing" TV show with Martin Sheen playing the US president. It is so easy to believe not only because we are used to wasteful spendings by government-funded bodies, but mostly because we have seen so many examples of inefficiency and misinterpreting real-world issues, adding enormous unnecessary complexity to proposed solutions and solving problems that don't exist.

This story is a myth. NASA also tried using pencils but decided to get rid of them due to issues of produced micro-dust, breaking tips, and potential flammability of wooden pencils. A private company Fisher Pen Company had developed what is now known as a "space pen" using their investments. Later, NASA tested the pen and decided to use it. The company also got an order from the Soviet Union and pens were sold across the ocean. The price for everyone was the same, $2.39 per pen.

Source: "Fact or Fiction?: NASA Spent Millions to Develop a Pen that Would Write in Space, whereas the Soviet Cosmonauts Used a Pencil" by Ciara Curtin, Scientific American

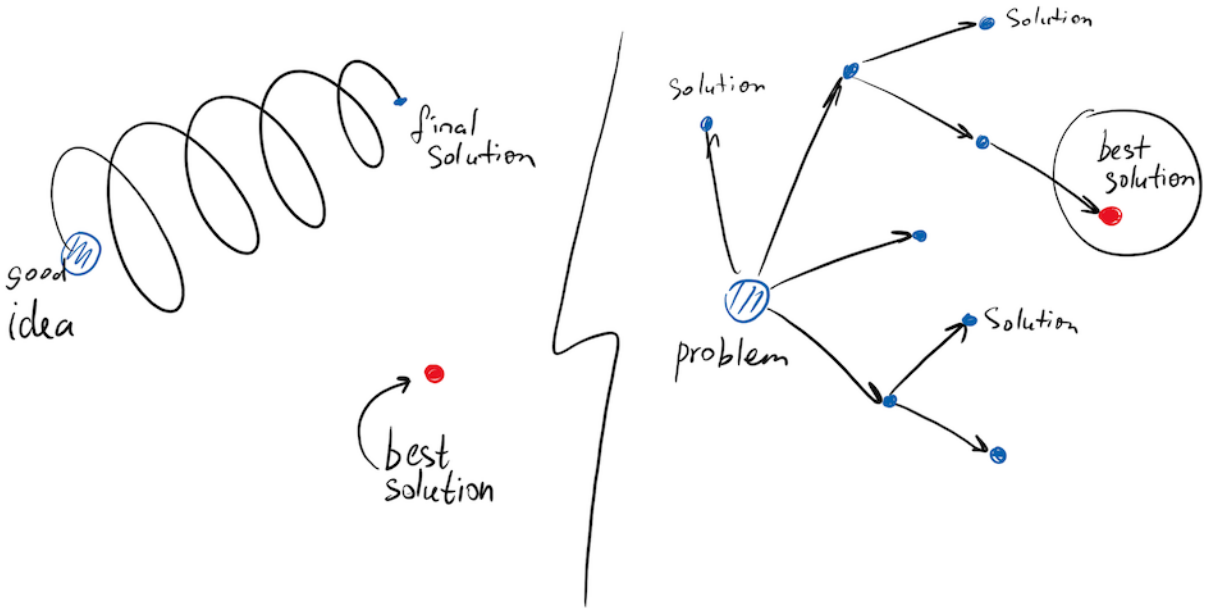
Fisher Space Pen

Here you can see the other part of the problem space versus solution space issue. Although the problem itself appeared to be simple, additional constraints, which we could also call *non-functional requirements*, made it more complicated than it looks like at first glance.

Jumping to a solution is very easy, and since each of us has a rather rich experience in solving everyday problems, we can find solutions for many issues almost immediately. However, as Bart Barthelemy and Candace Dalmagne-Rouge suggest in their article When You're Innovating, Resist Looking for Solutions[39] (2013, Harvard Business Review), thinking regarding solutions prevent our brain from keeping to think about the problem. Instead, we start going deeper into the solution that first came to our mind, adding more levels of details and making it more and more fit to be an ideal solution for a given problem.

One more aspect to consider when searching for a solution to a given problem. There is a danger of fixating all attention on one particular solution, which might be not the best one at all but came first to your mind, based on previous experiences, current understanding of the problem and other factors.

[39] https://hbr.org/2013/09/when-youre-innovating-resist-l



Refinement versus exploration

The exploratory approach to find and choose solutions involves more work spiking alternative ways to solve the problem, but the answer that is found during this type of exploration will most probably be much more precise and valuable. We will discuss more fixation on the first possible solution later in this chapter.

What went wrong with requirements

Many of us are familiar with the idea of requirements for software. Developers rarely have direct contact with the one who wants to solve some problem. Usually, some dedicated people such as requirements analysts, business analysis, or product managers, talk to customers and generalise outcomes of such conversations in the form of functional requirements.

Requirements can have different forms, from large documents called "requirements specification" to more "agile" means like user stories. Let's have a look at this example:

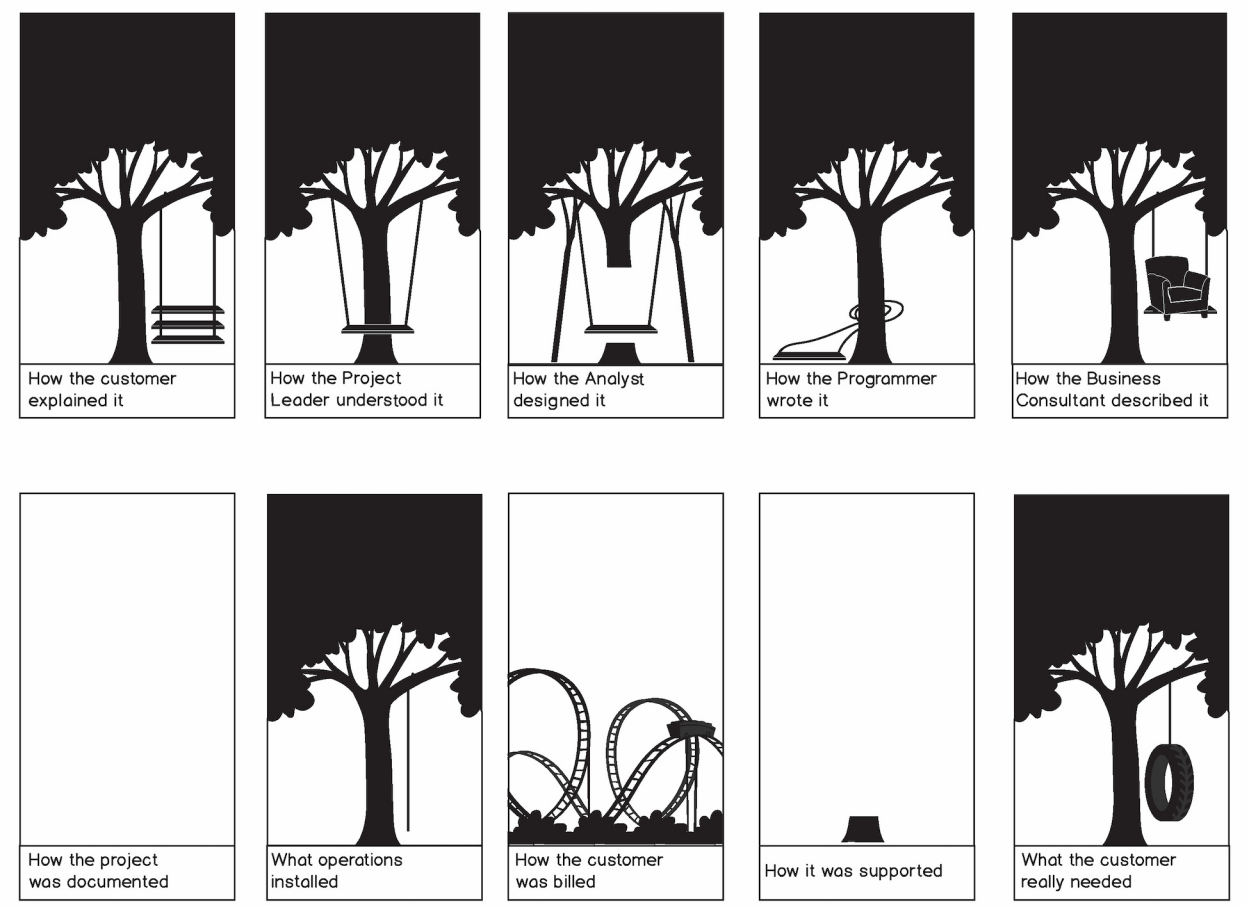
The system shall generate each day, for each hotel, a list of guests expected to check-in and check-out on that day.

As you can see, this statement only describes the solution. We cannot possibly know what the user is doing and what problem our system will be solving. Additional requirements might be specified, further refining the solution, but the problem description is never included in functional requirements.

In contrast, with user stories, we have more insight into what our user wants. Let's review this real-life user story: "*As a warehouse manager; I need to be able to print a stock level report; So I can order items when they are out of stock.*" However, this user story already dictates what developers need

to do. It is describing *the solution*. The real problem is probably that the customer needs a more efficient procurement process, so they never run out of stock. Alternatively, they need an advanced purchase forecasting system, so they can improve throughput without piling additional inventory in their warehouse.

Requirements became so notorious that if you search for an image using keywords "software requirements," the second result in Google Images would be this picture:



Tree Swing Project Management cartoon

We shall not think that requirements are waste. There are many excellent analysts out there, who produce high-quality requirements specifications. However, it is vital to understand, that these requirements are almost always represent the understanding of the actual problem by a person who wrote these requirements. A misconception that spending more and more time and money on writing higher quality requirements prevails in the industry.

However, lean and agile methodologies embrace more direct communication between developers and end users. Understanding the problem by everyone involved in building software, from end users to developers and testers, finding solutions together, eliminating assumptions, building prototypes for end users to evaluate - these things are being adopted by successful teams, and as we will see later in the book, they are also closely related to Domain-Driven Design.

Dealing with complexity

Complexity is something we are dealing with every day, consciously, and unconsciously. Merriam-Webster defines the word "complexity" as the quality or state of being complex. The world around us is somewhat chaotic, but we are using our instinct and experience to ignore this complexity or deal with it.

In software, the idea of complexity is not different. Most of the software is complex, as much as problems this software is trying to solve. Realising what kinds of complexity we are dealing with when creating software thus becomes very important.

Types of complexity

In 1986, the Turing Award winner Fred Brooks wrote a paper called *No Silver Bullet – Essence and Accident in Software Engineering* made a distinction between two types of complexity: essential and accidental complexity. Inherent complexity is coming from the domain, from the problem itself, and it cannot be removed without decreasing the scope of the problem. In contrast, accidental complexity is brought to the solution by the solution itself - this could be a framework, a database or some other infrastructure, different kinds of optimisation and integration.

Brooks argued that accidental complexity level decreased substantially during those years when the software industry became more mature. High-level programming languages and efficient tooling give programmers more time to work on business problems. However, as we can see today, more than thirty years later, the industry still struggles to fight accidental complexity. We will discuss some possible reasons for this phenomena in the next section.

You probably noticed that essential complexity has the strong relation to the problem space and accidental complexity leans towards the solution space. However, we often seem to get more complex problem statements than problems itself. Usually, this happens due to mixing problems with solutions, as we discussed before, or due to a lack of understanding.

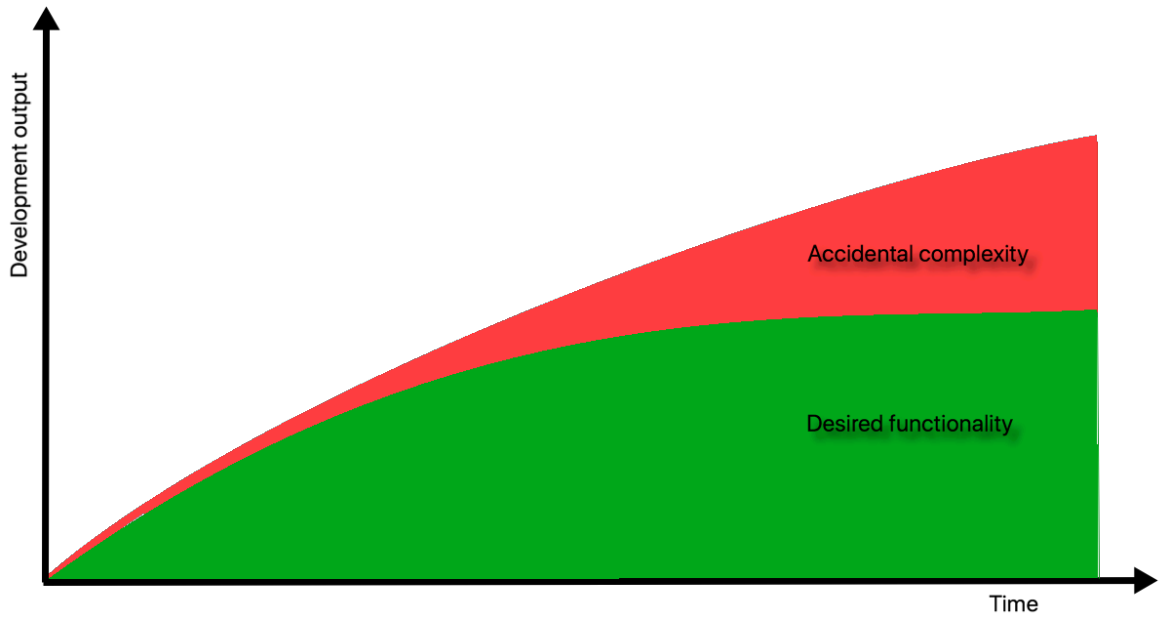
Gojko Adzic, a software delivery consultant, author of several influential books like *Specification by example* and *Impact Mapping*, is giving this example on his workshop:

The software-as-a-service company got a feature request to provide a particular report in real-time, which previously was executed once a month on schedule. After a few months of development, salespeople tried to get an estimated delivery date. The development department then reported that the feature would take at least six more months to deliver and the total cost would be around £1 million. It was because the data source for this report is in a transactional database and running it real-time would mean significant performance degradation, so additional measures like data replication, geographical distribution and sharding were required.

The company then decided to analyse the actual need that the customer, who requested this feature, had. It turned out that the customer wanted to perform the same operations as they were doing before, but instead of doing it monthly, they wanted it weekly. When asked about the desired

outcome of the whole feature, the customer then said that running the same report batched once a week would solve the problem. Re-scheduling the database job was by far easier operation that redesigning the whole system, while the impact for the end customer was the same.

This example clearly shows that not understanding the problem can lead to severe consequences. Developers tend to generalise and bring abstractions to solutions, and very often this is entirely unnecessary. What seems to be the essential complexity in this example, turned out to be a waste.



Complexity growth over time

The picture above shows that with the ever-growing complexity of the system, the essential part is being pushed down and the accidental part takes over. When systems become more prominent, a lot of effort is required to make the system work as a whole and to manage large data models, which large systems tend to have.

Domain-driven design helps you focus on solving complex domain problems and concentrates on the essential complexity. To do this, DDD offers several useful techniques for managing complexity by splitting the system into smaller parts and making these parts focused on solving a set of related problems. These techniques are described later in this book.

The rule of thumb when dealing with complexity is: embrace essential, or as we might call it, domain complexity and eliminate or decrease the accidental complexity. Your goal as a developer is not to bring too much accidental complexity. Hence that very often the accidental complexity is caused by over-engineering.

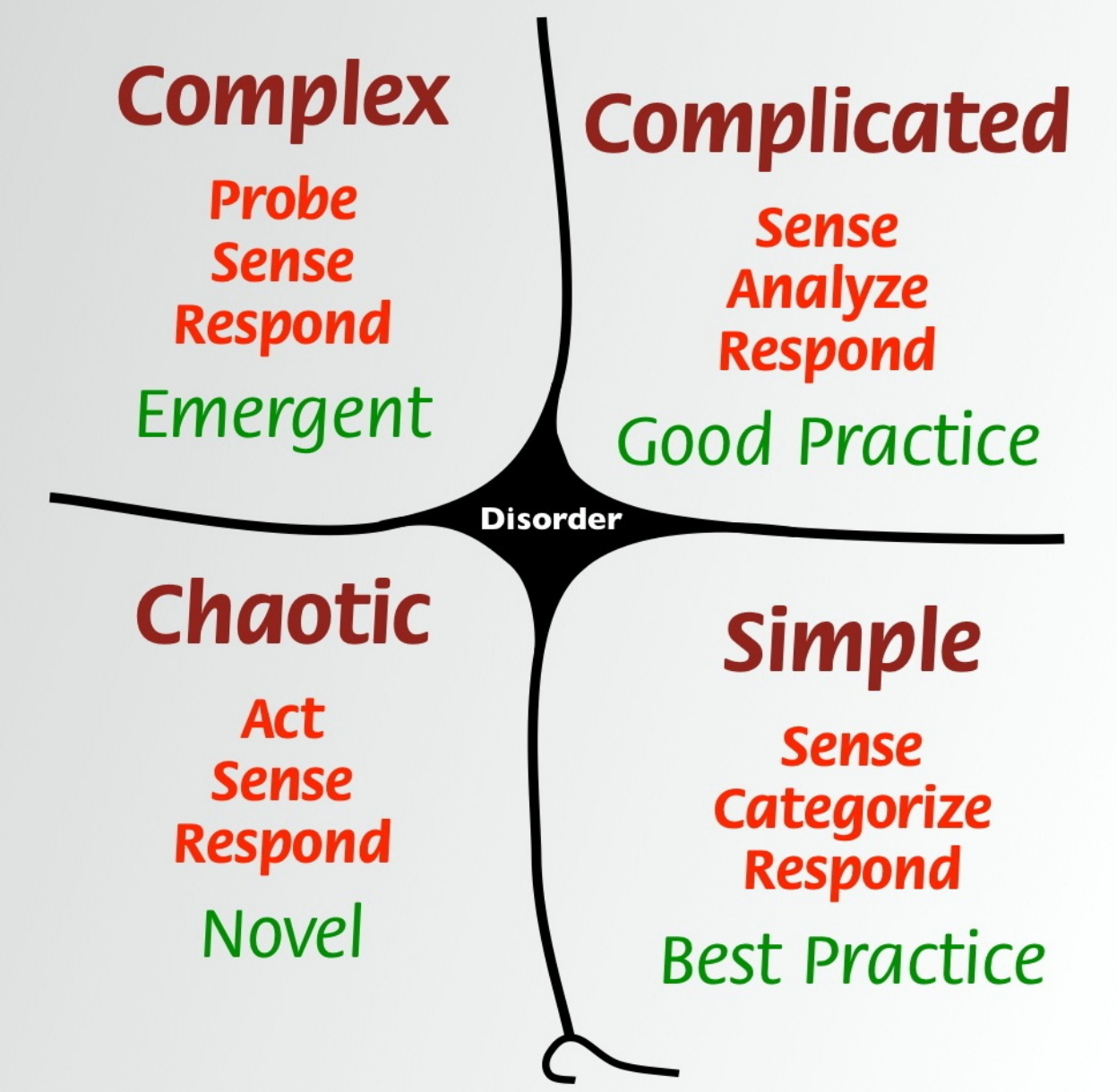
Categorising complexity

When dealing with problems, we don't always know if these problems were complex. Moreover, if they are complex, how complex? Is there a tool for measuring complexity? If there is, it would be beneficial to measure or at least categorise the problem complexity before starting to solve it. Such measurement would help to regulate the solution complexity as well, since complex problems also demand a complex solution, with rare exclusions from this rule. If you disagree, we will be getting deeper into this topic in the next section.

In 2007, Dave Snowden and Mary Boone published a paper *A Leader's Framework for Decision Making* in Harvard Business Review 2007. This paper won the "Outstanding Practitioner-Oriented Publication in OB" award from the Academy of Management's organisational behaviour division. What is so unique about it and which framework we describe there?

The framework is Cynefin. This word is Walsh for something like *habitat,* accustomed, familiar. Snowden started to work on it back in 1999 when he worked in IBM. The work was so valuable that IBM had established the Cynefin Centre for Organisational Complexity and Dave Snowden was its founder and director.

Cynefin divides all problems into five categories or complexity domains. By describing properties of problems that fall to each domain, it gives the *sense of place* for any given problem. After the problem is Categorised as one of the domains, Cynefin then also offers some practical approaches to deal with this kind of problem.



Cynefin Framework: image by Dave Snowden

These five domains have specific characteristics, and the framework provides both attributes for identifying to which domain your problem belong, and how the problem needs to be addressed.

The first domain is "simple," or "obvious." There you have problems, which can be described as *known knowns*, where best practices and an established set of rules are available, and there is a direct link between a cause and a consequence. The sequence of actions for this domain is *sense-Categorise-response*. Establishing facts (sense), identify processes and rules (Categorise) and execute them (response).

Snowden, however, warns about the tendency for people wrongly classify problems as "simple." He

identifies three cases for this:

- Oversimplification: this reason correlates with some of the cognitive biases described in the next section;
- Entrained thinking: when people blindly use the skills and experiences they obtained in the past and therefore become blinded to new ways of thinking;
- Complacency: when things go well people tend to relax and overestimate their abilities to react to the changing world. The danger of this case is because from being classified as "simple," such problems quickly escalate to "chaotic" domain due to a failure of people to adequately assess the risks.

For this book it is important to remember two main things:

- If you identify the problem as "obvious" - you probably don't want to set up a complex solution and perhaps would even consider buying some off-the-shelf software to solve the problem, if any software required at all.
- Beware, however, of wrongly classifying more complex problems in this domain to avoid applying wrong best practices instead of doing more thorough exploration and research.

The second domain is "complicated." There you find problems, where expertise and skills are required to find the relation between cause and effect since there is no single answer to such a problem. These are *known unknowns*. The sequence of actions in this domain would be *sense-analyse-respond*. As we can see, "analyse" replaced "Categorise" because there is no clear Categorisation of facts in this domain. Proper analysis needs to be done to identify, which good practice to apply. The classification can be used here too, but it requires to go through more choices and also some analysis of consequences needs to be done as well. That is where previous experience is necessary. Engineering problems are typically in this category when clearly understood problem requires the more sophisticated technical solution.

In this domain, assigning qualified people to do some design up front and then perform the implementation makes perfect sense. When a thorough analysis is done, the risk of implementation failure is low. Here it makes sense to apply DDD patterns for both strategic and tactical design, and to the implementation, but you probably could avoid more advanced exploratory techniques like event-sourcing. Also, you might spend less time on knowledge crunching, if the problem is thoroughly understood.

"Complex" is the third complexity domain in Cynefin. Here we encounter something that no one has done before. Making even a rough estimate is impossible. It is hard or impossible to predict the reaction in response to our action, and we can only find out about the impact that we have made in retrospect. The sequence of actions in the "complex" domain is "probe-sense-respond." There are no right answers here and no practices to rely upon the previous experience might not be helping. These are *unknown unknowns*, and this is a place where all innovation happens. Here we find our Core Domain, the concept, which we will get to later in the book.

The course of actions for the "complex" domain is lead by experiments and spikes. There is very little sense to make a big design upfront since we have no idea, how stuff will work and how the world would react to what we are doing. Work here needs to be done in small iterations with continuous and intensive feedback.

Advanced modelling and implementation techniques that are lean enough to allow to respond to changes quickly are the perfect fit in this domain. In particular, modelling using EventStorming and implementation using event-sourcing are very much at home in the "complex" area. The thorough strategic design is necessary, but some tactical patterns of DDD can be safely ignored when doing spikes and prototypes, to save time. However, again, event-sourcing could be your best friend. Both EventStorming and event-sourcing are described later in the book.

Fourth domain is "chaotic". Here is where hellfire burns and the Earth spins faster than it should. No one wants to be in here. Appropriate actions here would be *act-sense-respond* since there is no time for spikes. It is probably not the best place for DDD since there is no time and fiscal budget for any sort of design available at this stage.

"Disorder" is the fifth and final domain, right in the middle. It is where the transition to chaos usually happens from any stage. Underestimated complex problems with unrealistic deadlines bring teams to stress, leading to disorder at the later project stages, from where all slips to chaos.

Here we only have a brief overview of the complexity classification. There is more to it, but for this book, the most important outcome is that DDD can be applied almost everywhere, but it is virtually of no use in obvious and chaotic domains. EventStorming as a design technique for complex systems would be useful for both complicated and complex domains, along with event-sourcing, which suits the complex domain best.

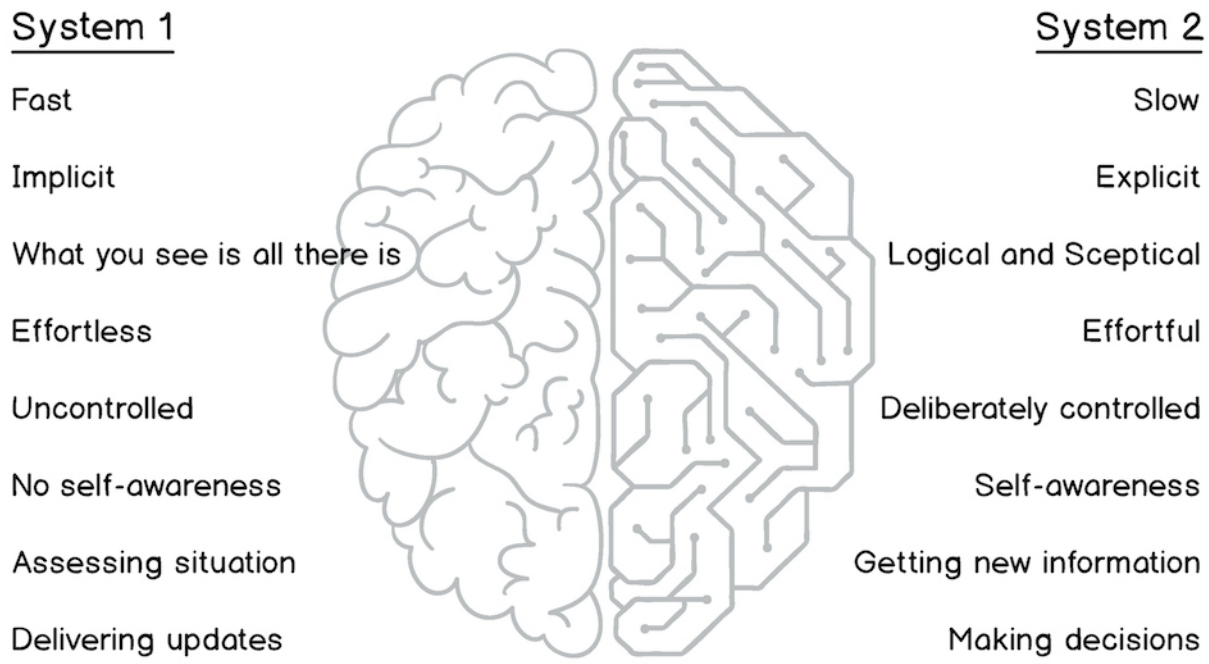
Decision making and biases

The human brain processes a tremendous amount of information every single second. We do many things on some autopilot, driven by instincts and habits. Most of our daily routines are like this. Another area of brain activity is thinking, learning and decision making. Such actions are being performed significantly slower and require much more power than those "automatic" operations.

Dual process theory in psychology suggests that these types of brain activity are indeed entirely different and there are two different processed for two kinds of thinking. One is the implicit, automatic, unconscious process, and the other one is an explicit conscious process. Unconscious processes are formed for a long time and also very hard to change since changing such a process would require developing a new habit, and this is not an easy task. The conscious process, however, can be altered through logical reasoning and education.

These processes, or *systems*, happily co-exist in one brain, but are rather different in ways how they operate. Keith Stanovich and Richard West have coined the names *implicit system*, or *System 1* and *explicit system*, or *System 2* (*Individual difference in reasoning: implications for the rationality debate*?. Behavioural and Brain Sciences 2000). Daniel Kahneman in his award-winning book

Thinking Fast and Slow (New York: Farrar, Straus and Giroux, 2011) assigned several attributes to each system:



System 1 and System 2

What all this has to do with Domain-Driven Design? Well, the point here is more about how we make decisions. The Cynefin complexity model requires from us at least to Categorise the complexity we are dealing with in our problem space (and also sometimes in the solution space). But to assign the right category, we need to make a lot of decisions, and here we often get our *System 1* speaking and making assumptions based on many of our biases and experiences from the past, rather than engaging the *System 2* to start reasoning and thinking. Of course, every one of us is familiar with a colleague exclaiming "yeah, that's easy!" before you can even finish describing the problem. We also often see people organising endless meetings and conference calls to discuss something that we assume to be a straightforward decision to make.

Cognitive biases are playing a crucial role here. Some biases can profoundly influence the decision making and this is definitely "the system 1 speaking". Here are some of the biases and heuristics that can affect your thinking about the system design, which you can recognise:

- **Choice-supportive bias**: If you have chosen a thing, you will positive about this choice despite your choice might have been proven to contain significant flaws. Typically it happens when we get to the first model and try to stick to it at all costs although it becomes evident that the model is not optimal and needs to be changed. Also, such bias can be observed when one chooses a technology to use, like a database or a framework. Despite many arguments against using these the preferred technique, it will be tough to make something that left a trace in the

heart.

- **Confirmation bias**: Very close to the previous one, the confirmation bias makes you only to hear arguments that support your choice or position and ignore arguments that contradict your views on it, although these arguments may show that your opinion is wrong.
- **Band-wagon effect**: When the majority of people in the room agree on something, this "something" begins to make more sense to the minority that previously disagreed. Without engaging the *System 2*, the opinion of the majority gets more credit without any objective reason. Remember that what the majority decides is not the best choice by default!
- **Overconfidence**: Too often people tend to be too optimistic about their abilities. This bias might cause them to take more significant risks and take wrong decisions that have no objective grounds but based exclusively on their opinion. The most obvious example of this is the estimation process. It happens much more often when people underestimate the time and effort they are going to spend on some problem than overestimate.
- **Availability heuristic**: The information in hand is not always, or, *always not* all information, which we can get about a particular problem. People tend to base their decisions only with information in hand, without even trying to get more details. Too often this leads to over-simplification of the domain problem and underestimation of the essential complexity. This heuristic can also trick us when we make technological decisions and chose something that "always worked" without analysing operational requirements, which might be much higher than the technology of our choice can handle.

The importance of knowing how our decision-making process works is hard to overestimate. The books referenced in this section contain much more information about human behaviour and different factors that can have a negative impact on our cognitive abilities. We need to remember to turn on the *System 2* in order to make better decisions, which are not based on emotions and biases.

Knowledge

Many junior developers tend to think that software development is typing code and when they become more experienced in typing, will know more IDE shortcuts and learn frameworks and libraries by heart - they will be ninja developers, being able to write something like Instagram in a couple of days.

Well, the reality is harshly different. In fact, after getting some experience and after deliberately spending months and maybe years in death-marches towards impossible deadlines and unrealistic goals, people usually slow down. They begin to understand that writing code immediately after receiving some specification might not be a perfect idea. The reasons for this might be already apparent to you after you have read all previous sections. Being obsessed with solutions instead of understanding the problem, ignoring essential complexity and conforming biases - all these factors influence us when we are developing software. As soon as we get more experience and learn on our own mistakes and, preferably, on errors of the others, we realise that the most crucial part of writing useful, valuable software is the knowledge.

> Software development is a learning process. Working software is a side-effect. *Alberto Brandolini*

Domain knowledge

Not all knowledge is equally useful when building a software system. Knowing about writing Java code in the financial domain might not be very beneficial when you start creating an iOS app for real estate management. Of course, principles like Clean Code, DRY and so on are helpful no matter what programming language you use. However, the business knowledge of one domain might be vastly different from what you need for some other domain.

That is where we encounter the concept of domain knowledge. Domain knowledge is the knowledge about the domain where you are going to operate with your software. If you are building a trading system - your domain is financial trading, and you need to gain some knowledge about trading to understand what your users are talking about and what they want.

It all comes to getting into the problem space. If you are not able to at least understand the terminology of the problem space - it would be hard if not impossible even to speak to your future users. If you lack the domain knowledge, the only source of information for you would be the "specification". When you have at least some domain knowledge - conversations with your users become more fruitful since you begin to understand what people are talking about. One of the consequences of that would be building trust between a customer and a developer. Such confidence is hard to overestimate. A trusted person gets more insight and mistakes are forgiven easier. By speaking the *domain language* to *domain experts* (your users and customers), you also gain credibility, and they see you and your colleagues as more competent people.

Obtaining the domain knowledge is not an easy task. People specialise in their domains for years and decades, they become experts in their domains, and they do this kind of work for a living. Software developers and business analysts do something else, and that particular problem domain might be little known or completely unknown at the moment they start to obtain the domain knowledge.

The art of obtaining the domain knowledge is effective collaboration. Domain experts are the source of ultimate truth, at least we want to treat them like this. However, they might not be. Some organisations have this knowledge fragmented; some might have it just wrong. Knowledge crunching in such environments is even harder, but there might be bits and pieces of information, waiting to be found at desks of some low-level clerks and your task would be to see it.

The general advice here is to talk to people and talk to many different people, from inside the domain, from the management of the whole organisation and adjacent domains. There are several techniques to obtain the domain knowledge and here are some of them:

- Conversations are the most popular method, formalised as meetings. However, conversations often turn into a mess without any visible outcome. Still, some value is there, but you need to listen carefully and ask many questions to get valuable information.

- Observation is a very powerful technique, which heavily correlates with the Lean Startup motto *Get out of the building*. Software people need to fight their introversion, leave the ivory tower and go to a trading floor, to a warehouse, to a hotel, to a place where business runs, and then talk to people and see how they work. Jeff Patton gave many good examples in his talk at DDD Exchange 2017[40].
- Domain Story-Telling[41], a technique proposed by Stefan Hofer and his colleagues from Hamburg University advocates using pictograms, arrows and a little bit of text, plus numbering actions sequentially, to describe different interactions inside the domain. The technique is easy to use, and typically there is not much to explain to people participating in such a workshop before they start using it to deliver the knowledge.
- EventStorming, an advantageous technique, which is invented and coined by Alberto Brandolini. He explains the method in his book *Introducing EventStorming* (2017, Leanpub) and we will also go into more details later in this book when we start analysing our sample domain. EventStorming uses post-it notes and a paper roll to model all kinds of activities in a straightforward fashion. Workshop participants write facts of the past (events) on post-its and put them on the wall, trying to make a timeline. It allows discovering activities, workflows, business process and so on. Very often it also uncovers ambiguities, assumptions, implicit terminology, confusion and sometimes conflicts and anger. In short - everything, what the domain knowledge consists of.

Avoiding ignorance

Ignorance is the single greatest impediment to throughput. - Dan North

Back in 2000, Philip Armour published an article called *Five orders of ignorance* (Communications of the ACM, Volume 43 Issue 10, Oct. 2000), with a subtitle *Viewing software development as knowledge acquisition and ignorance reduction*. This message very much correlates with Alberto's quote from the previous section, although it is somewhat less catchy but by no mean less powerful. The article argues that increasing domain knowledge and decreasing ignorance are two keys to creating software that delivers value.

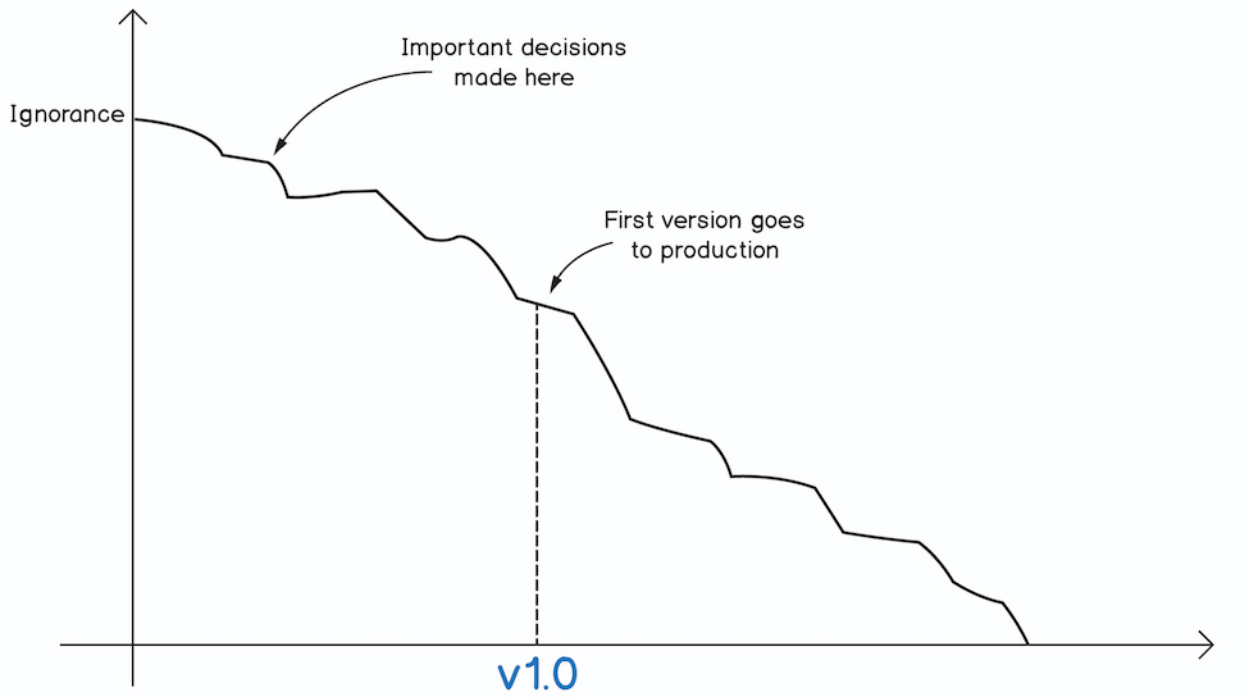
The article concentrates on ignorance and identifies five levels of it:

1. Zero ignorance level is the lowest. On this level, you have no ignorance since you got most knowledge and knew what to do, and how to do it.
2. The first level is when you don't know something, but you realise and accept this fact. You want to get more knowledge and decrease ignorance to level zero, so you have channels to obtain the knowledge.
3. The second level is when you don't know that you don't know. Most commonly this occurs when you get a specification that describes a solution without specifying, which problem this solution is trying to solve. This level can also be observed when people pretend to have

[40] https://skillsmatter.com/skillscasts/10127-empathy-driven-design
[41] http://domainstorytelling.org/

competence they do not possess, and at the same time being ignorant of this. Such people might be lacking both business and technical knowledge. A lot of wrong decisions are made at this level of ignorance.
4. The third level is when you don't even know how to find out that you don't know something. It is tough to do anything on this level since apparently there is no way to access end customer even to ask if you understand their problem or not, to get down to level two. Building a system might be the only choice in this case, since it will be the only way to get any feedback.
5. The fifth and the last level of ignorance is meta-ignorance. It is when you don't know about five degrees of ignorance.

As you can see, ignorance is the opposite of knowledge. The only way to decrease ignorance is to increase understanding. High level of ignorance, conscious or subconscious, leading to the lack of knowledge and misinterpretation of the problem, and therefore, increasing the change of building a wrong solution.



Ignorance is highest at the earliest stages

Eric Evans, the father of DDD, describes the upfront design as *locking in our ignorance*. The issue with the upfront design is that we do it at the beginning of a project. At that time we have the least knowledge and most ignorance. It became a norm to make most important decisions about the design and architecture of the software at the very beginning of projects when there is virtually no ground for such decisions available. This practice is quite obviously not optimal. Mary and Tom Poppendieck in their book *Lean Software Development* (2003, Addison-Wesley Professional) described the term "last responsible moment" as "the moment at which failing to make a decision eliminate an important alternative" and, as the term suggests, advise to postpone important decisions.

Agile and Lean methodologies also help to communicate knowledge more efficient and awareness of ignorance more obvious.

In the article Introducing Deliberate Discovery[42], Dan North suggests that we realise our position of being on at least the second level of ignorance when we start any project. In particular, the following three risks need to be taken into account:

- A few Unpredictable Bad Things will happen during the project
- Being Unpredictable, these Things are unknown in advance
- Being Bad, these Things will negatively impact the project

To mitigate these risks, Dan recommends using *deliberate discovery*, i.e., seeking knowledge from the start. Since not all knowledge is equally important, we need to try identifying those sensitive areas, where ignorance is creating most impediments. By raising knowledge levels in these areas, we enable progress. At the same time, we need to keep new troublesome areas and resolve them too; and this process is continuous and iterative.

Summary

Make no mistake by thinking that you can deliver valuable solutions to your customers just by writing code. And that you can deliver faster and better by typing more characters per second and writing cleaner code. Customers do not care about your core or how fast you type. They only care that your software solves their problems in a way that no one solved it before. As Gojko Adžić wrote in his sweet little book about impact mapping (*Impact Mapping: Making a Big Impact With Software Products and Projects,* 2012Provoking Thoughts), you cannot only formulate user stories like:

- As a *someone*
- To *do something*
- I need to *use some functionality*

Your user *someone* might be already doing *something* by executing *some functionality* even without your software. Using a pen and paper. Using Excel. Using a system of your competitor. What you need to ensure is that you make a difference, make an impact. Your system will let people work faster, more efficient, allow them to save money or even not to do this work at all if you completely automate it.

To build such software, you must understand the problem of your user. You need to crunch the domain knowledge, decrease the level of ignorance, accurately classify the problem complexity and try to avoid cognitive biases on the way to your goal. This is an essential part of Domain-Driven Design, although not all of these topics are covered in the Blue Book.

[42] https://dannorth.net/2010/08/30/introducing-deliberate-discovery/

Further reading

- Snowden D J, Boone M E. (2007). “A leader’s framework for decision making”. Harvard Business Review 2007 November issue
- Kahneman, Daniel (2011). “Thinking, fast and slow” (1st ed.). New York: Farrar, Straus, and Giroux.
- Adžić, G. (2012). “Impact Mapping: Making a Big Impact With Software Products and Projects”. Provoking Thoughts.

Multiple Canonical Models – Martin Fowler

Multiple Canonical Models

Adapted from an article published in 2003

Scratch any large enterprise and you'll usually find some kind of group focused on enterprise-wide conceptual modeling. Most commonly this will be a data management group, occasionally they may be involved in defining enterprise-wide services. They are enterprise-wide because rather than focusing on the efforts of a single application they concentrate on integrating multiple applications.

Most such groups tend to focus on creating a single comprehensive enterprise model. The idea is that if all applications operate based on this single model, then it will be much easier to integrate data across the whole enterprise - thus avoiding stovepipe applications. Much of this thinking follows the shared database approach to enterprise integration - where integration occurs through applications sharing a single logical enterprise-wide database.

A single conceptual model is a tricky beast to work with. For a start it's very hard to do one well - I've run into few people who can build these things. Even when you've built one, it's hard for others to understand. Many times I've run into the complaint that while a model is really good - hardly anyone understands it. This is, I believe, an essential problem. Any large enterprise needs a model that is either very large, or abstract, or both. And largeness and abstractness both imply comprehension difficulties.

These days many integration groups question the shared database approach, instead preferring a messaging based approach to integration. I tend to agree with this view, on the basis that while it's not the best approach in theory, it better recognizes the practical problems of integration - especially the political problems.

One of the interesting consequences of a messaging based approach to integration is that there is no longer a need for a single conceptual model to underpin the integration effort. Talking with my colleague Bill Hegerty I realized that

- You can have several canonical models rather than just one.
- These models may overlap
- Overlaps between models need not share the same structure, although there should be a translation between the parts of models that overlap
- The models need not cover everything that can be represented, they only need to cover everything that needs to be communicated between applications.

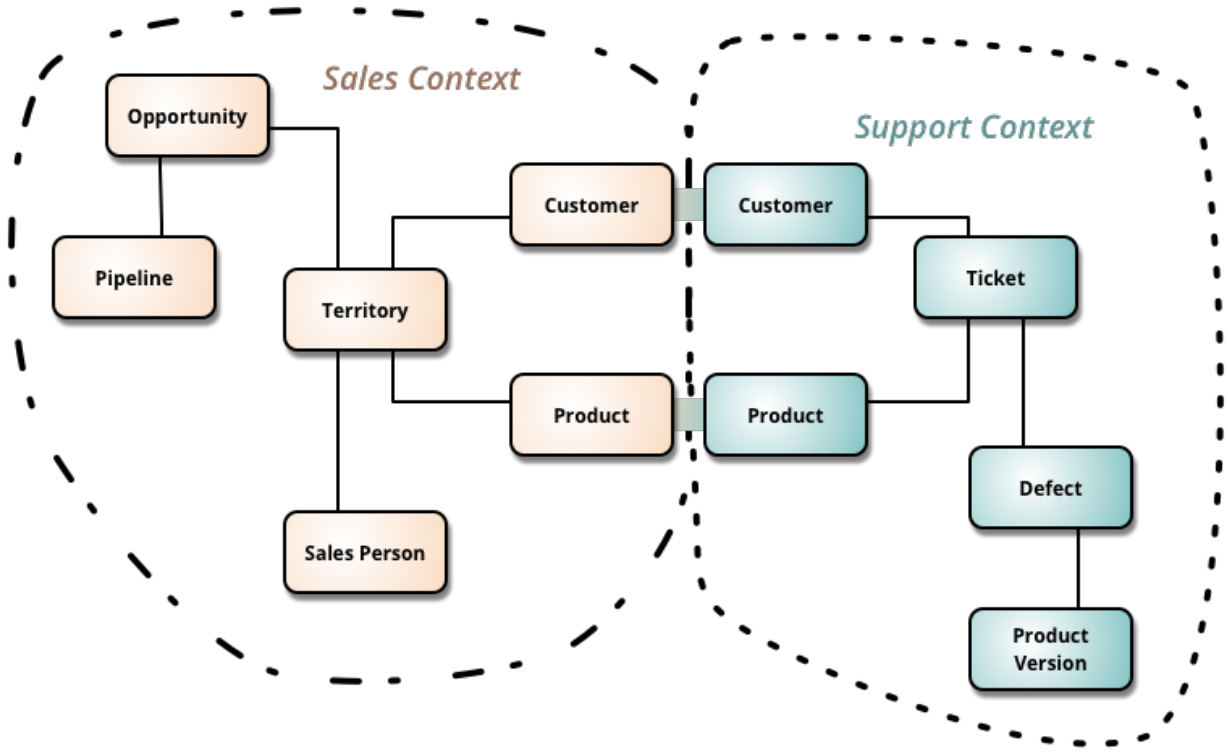
- These models can be built through harvesting, rather than planned up-front. As multiple applications communicate pair-wise, you can introduce a canonical model to replace n * n translation paths with n paths translating to the canonical hub.
- The result breaks down the modeling problem, and I believe simplifies it both technically and politically.

So far, however, it seems that the data modeling community is only beginning to catch on to this new world. This is sad because data modelers have a tremendous amount to offer to people building canonical messaging models. Not just are skills not taking part, many also resist this approach because they assert that a single enterprise-wide model is the only proper foundation for integration.

Bounded Contexts

Adapted from an article published in 2014

Bounded Context is a central pattern in Domain-Driven Design. It is the focus of DDD's strategic design section which is all about dealing with large models and teams. DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.



A diagram of Context Map

DDD is about designing software based on models of the underlying domain. A model acts as a UbiquitousLanguage[43] to help communication between software developers and domain experts. It also acts as the conceptual foundation for the design of the software itself - how it's broken down into objects or functions. To be effective, a model needs to be unified - that is to be internally consistent so that there are no contradictions within it.

As you try to model a larger domain, it gets progressively harder to build a single unified model. Different groups of people will use subtly different vocabularies in different parts of a large organization. The precision of modeling rapidly runs into this, often leading to a lot of confusion. Typically this confusion focuses on the central concepts of the domain. Early in my career I worked with a electricity utility - here the word "meter" meant subtly different things to different parts of the organization: was it the connection between the grid and a location, the grid and a customer, the physical meter itself (which could be replaced if faulty). These subtle polysemes[44] could be smoothed over in conversation but not in the precise world of computers. Time and time again I see this confusion recur with polysemes like "Customer" and "Product".

In those younger days we were advised to build a unified model of the entire business, but DDD recognizes that we've learned that "total unification of the domain model for a large system will not be feasible or cost-effective" [1]. So instead DDD divides up a large system into Bounded Contexts, each of which can have a unified model - essentially a way of structuring multiple canonical models.

Bounded Contexts have both unrelated concepts (such as a support ticket only existing in a customer support context) but also share concepts (such as products and customers). Different contexts may have completely different models of common concepts with mechanisms to map between these polysemic concepts for integration. Several DDD patterns explore alternative relationships between contexts.

Various factors draw boundaries between contexts. Usually the dominant one is human culture, since models act as Ubiquitous Language, you need a different model when the language changes. You also find multiple contexts within the same domain context, such as the separation between in-memory and relational database models in a single application. This boundary is set by the different way we represent models.

DDD's strategic design goes on to describe a variety of ways that you have relationships between Bounded Contexts. It's usually worthwhile to depict these using a context map.

Further Reading

- The canonical source for DDD is Eric Evans's book[45]. It isn't the easiest read in the software literature, but it's one of those books that amply repays a substantial investment. Bounded Context opens part IV (Strategic Design).

[43] https://martinfowler.com/bliki/UbiquitousLanguage.html
[44] http://en.wikipedia.org/wiki/Polysemy
[45] https://amzn.to/2AUG3q0

- Vaughn Vernon's Implementing Domain-Driven Design[46] focuses on strategic design from the outset. Chapter 2 talks in detail about how a domain is divided into Bounded Contexts and Chapter 3 is the best source on drawing context maps.
- I love software books that are both old and still-relevant. One of my favorite such books is William Kent's Data and Reality[47]. I still remember his short description of the polyseme of Oil Wells.
- Eric Evans describes how an explicit use of a bounded context can allow teams to graft new functionality in legacy systems using a bubble context[48]. The example illustrates how related Bounded Contexts have similar yet distinct models and how you can map between them.

[46] https://amzn.to/2MwG58S
[47] https://amzn.to/2vutD3c
[48] http://domainlanguage.com/wp-content/uploads/2016/04/GettingStartedWithDDDWhenSurroundedByLegacySystemsV1.pdf

From Human Decisions, to Suggestions to Automated Decisions – Jef Claes

Published on jefclaes.be 2017-07-01

I help out during the weekends in a small family run magic shop. I'm the third generation working in the shop. My great-grandfather always hoped that his only son would follow in his footsteps as a carpenter. But at only eighteen years old, my grandfather said goodbye to the chisels and sawdust, and set out for the big city to chase his dream of becoming a world class magician. The first few years were tough, he was no Houdini. He would (hardly) get by performing at kid birthday parties, weddings and store openings. That's how he met my late grandmother. She worked as a shop girl in one of the first malls that were built in the city, and happened to show up each time my grandfather performed in one of the stores. After getting married, having a baby (my dad) and saving every dime they earned, my grandfather was able to rent a hole in the wall and open up his own tiny magic shop - in that same mall. Once my dad finished school, he worked as a middle school teacher for a few years, giving up on that job to join his father in the family business. He loves to tell you how he can now still teach children, without the chore of grading their homework. I've been running around and helping out in the store since I could barely walk. I guess you can say that magic runs in our blood.

Since the beginning of time, our trade has relied on secrecy. However, due to the rise of the internet, magic is dying a slow death. Even the greatest of tricks and illusions are challenged and destroyed in the open by non-believers. Our craft is now reduced by many to a cheap fairground attraction.

My grandfather, even after suddenly losing grandma last year, isn't willing to give up on the business though. "There will be a time when the people need magic once more, and we will be waiting right here." He decided not to see modern technology as the nemesis of magic, but rather as a potential assistant.

Instead of fiddling in his study all night with a book of cards, a hat, a scarf and the lonely rabbit, I've been working with him trying to show him what happened in technology over the last 30 years. Being a programmer, I started by showing off some of my unfinished hobby projects experimenting with micro controllers. I hadn't gotten much further than making some LEDs blink controlled by my voice, but that was enough to spark my grandfather's creativity. "Can that chip make the lights go out? Can it blow smoke? Can it sense if I flip it around real fast?" One evening, tired after brainstorming and testing ideas all night, he told me "being able to tell these little computers what to do might not be magic, but a miracle".

I would like to tell you the details of what we came up with, but I'd have to make you disappear after I did. To make a long story short, it was an overwhelming success. Neighbourhood magicians

picked it up, and even mere mortals thought it was a great gimmick they could show off with. A friend of mine even told me he used our invention to pick up girls. It wasn't for long before I got daily emails and tweets from all over the world, begging me to ship our gadget their way.

Thanks to some lovely open source software, I was able to set up a full blown web shop in a matter of days. While orders started rolling in, we got a grip on how to actually produce our new product at a sufficient pace. Shipping overseas turned out to be surprisingly easy. What we didn't anticipate for was how to handle returns and refunds. This is where the actual story starts…

When our usual customers visit the store, we take our sweet time to show them how to perform the trick. This results in us knowing our customers pretty well and hardly ever having anyone return an item or ask for a refund. Admittedly, growing the same connection with our customers online hasn't been a success. A lot of them lack the magician mindset. You can't just buy magic, you have to put the practice in to make the magic happen. These maladjusted expectations make for quite a few phoney complaints.

Me, my dad and my grandfather in turn have been performing the chore of handling returns and refunds. This domain of the open source shop is very much underdeveloped. Even something as simple as looking up a customer's details and order history, requires scanning multiple pages of information or even querying the database by hand. Making a well-informed decision takes up way more time than it should.

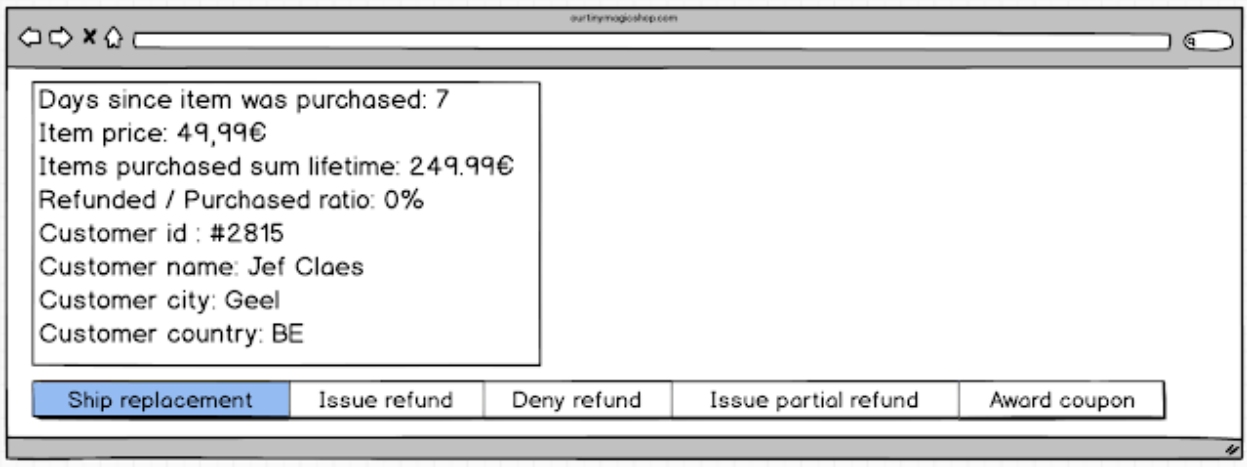
A use case specific view

The first thing I did in an attempt to speed up this process was building a use case specific view. I asked my dad and grandfather which heuristics they use and which data is needed to feed those heuristics. To get the full picture as soon as possible, I imposed the rule that only this specific view can be used to make a decision. If a piece of data was lacking, I would add it the same day.

This process was more useful than I expected. What we learned is that we all used different heuristics, but were also victim to different biases. For example, I learned that my grandfather used to have a Dutch neighbour who would leave for work very early, and slam the door so loud, it woke my grandfather up each morning. He has grown a disliking for the Dutch ever since. When customers were Belgian, he would much more lean towards issuing a refund, since he believes Belgians are less likely to lie about the cause of a broken item. We also discovered that we used different words for specific numbers. I would use "Items purchased sum", but my dad would use "Items purchased lifetime" to define the total amount of money spent purchasing items. We decided on being more explicit and making a composition of all those words.

I ended up with a simple screen that rendered a read model that looked something like this.

```
type ReadModel = {
    DaysSinceItemPurchased : int
    ItemPrice : decimal
    ItemPurchasedSumLifetime : decimal
    RefundedToPurchasedItemsRatio : decimal
    CustomerId : string
    CustomerName : string
    CustomerCity : string
    CustomerCountryCode : string
}
```



Readmodel

Based on heuristics in the head and a snapshot of information available in the world, we would make a decision and click a button to execute a specific command.

```
type Command =
    | IssueRefund
    | IssuePartialRefund
    | AwardCoupon
    | ShipReplacement
    | DenyRefund
    | DeferDecision

type Snapshot = {
    SnapshotId : string
    Value : ReadModel
}

type Decision = {
    BasedOnSnapshotId : string
```

```
    Command : Command
    MadeBy : string
}

type MakeDecision = Snapshot -> Decision

let makeHumanDecision by : MakeDecision =
    // Heuristics in the head, fed by knowledge from the world
    fun snapshot ->
        {
            BasedOnSnapshotId = snapshot.SnapshotId
            Command = IssueRefund
            MadeBy = by
        }
```

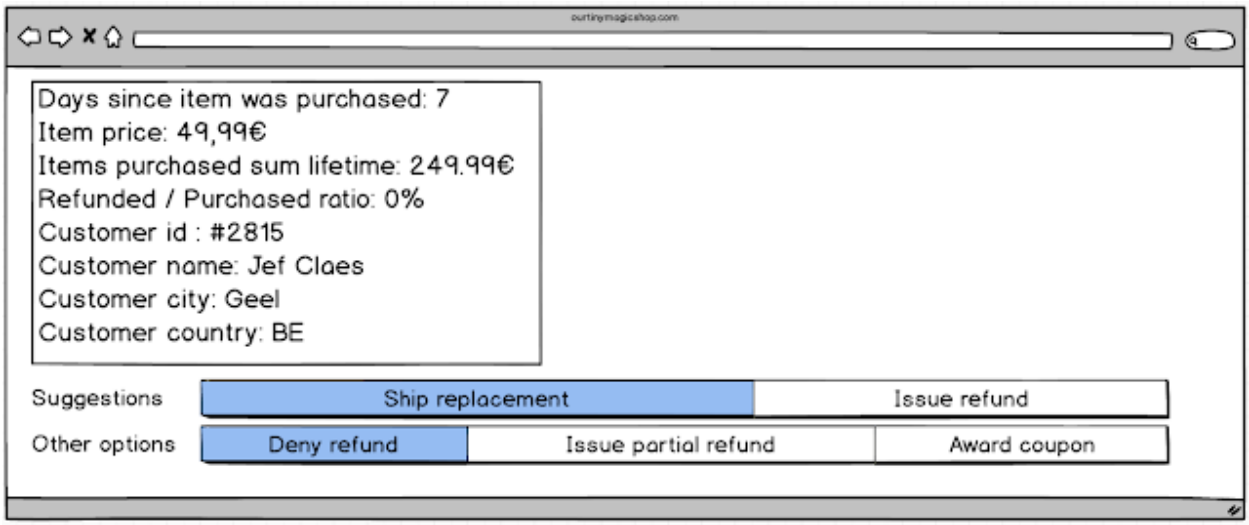
Making suggestions

By now, I had gotten quite interested in this domain. Each day I would go over all of the decisions and see whether I understood why a decision was made. I would call the shop each time I didn't understand and scribble down notes whenever I discovered a new implicit rule. In the meanwhile, I started experimenting with codifying these rules to make automated decisions, but when I ran the older snapshots through my routine the results were not 100% there yet. Instead of making automated decisions, I switched to making suggestions instead.

```
type Suggestion = {
    Command : Command
    Confidence : int
}

type Suggestions = {
    BasedOnSnapshotId : string
    Options : Suggestion seq
}

type MakeSuggestion = Snapshot -> Suggestions

let machineMakesSuggestions snapshot =
    let returnWindowInDays = 14
    let itemInReturnWindow = snapshot.Value.DaysSinceItemPurchased <= returnWindowIn\
Days

```

```
18      let options =
19          if itemInReturnWindow then
20              [
21                  { Command = ShipReplacement; Confidence = 51 }
22                  { Command = IssueRefund; Confidence = 49 }
23                  { Command = DenyRefund; Confidence = 0 }
24                  { Command = AwardCoupon; Confidence = 0 }
25                  { Command = IssuePartialRefund; Confidence = 0 }
26              ]
27          else
28              [
29                  { Command = DenyRefund; Confidence = 50 }
30                  { Command = AwardCoupon; Confidence = 30 }
31                  { Command = IssuePartialRefund; Confidence = 10 }
32                  { Command = ShipReplacement; Confidence = 10 }
33                  { Command = IssueRefund; Confidence = 0 }
34              ]
35
36      { BasedOnSnapshotId = snapshot.SnapshotId; Options = options }
```

I rendered these suggestions on top of the existing view and observed the decisions that were made.



Suggestions

```
type PickSuggestion = Suggestions -> Decision

let humanPicksSuggestion by : PickSuggestion =
    fun suggestions ->
        {
            BasedOnSnapshotId = suggestions.BasedOnSnapshotId
            Command = IssueRefund
            MadeBy = by
        }
```

My partners in crime were quite ecstatic with this new feature. They had a bit more room to breathe and could spend more time doing things they liked.

After observing and comparing the suggestions with the decisions made, I kept tweaking the routine a bit more. I got close but I felt as if I wasn't quite there yet.

Automating decisions

It was my grandfather who eventually pushed me to fully automate making these decisions. He said "I almost always find myself picking the first option. It's fine if the machine is a bit off now and then. Just defer decisions and call in a human when the machine is not confident enough." How can I question my grandfather's wisdom? And so this happened...

```
let machinePicksSuggestion : PickSuggestion =
    fun suggestions ->
        let confidenceTreshold = 50

        let command =
            suggestions.Options
            |> Seq.filter (fun x -> x.Confidence >= confidenceTreshold)
            |> Seq.sortByDescending (fun x -> x.Confidence)
            |> Seq.map (fun x -> x.Command)
            |> withFallbackTo DeferDecision
            |> Seq.head

        {
            BasedOnSnapshotId =  suggestions.BasedOnSnapshotId
            Command = command
            MadeBy = "Machine"
        }
```

Ta-da!

With that, we've come full circle. I'm happy to report that my dad, grandfather and I are back to spending more time in my grandfather's study coming up with new tricks.

I regularly have a look at the data to check for anomalies and to tweak the routine a bit further. But even that is taking up less and less time. To be fair, I didn't invest in a full blown test suite even though this small routine has grown into 200 lines of code. I find much relief in the fact that when I replay past decisions, I hardly ever find a regression.

Maybe if we invent another popular trick like this one, we will acquire enough data to let the machine do the learning for me. For now, it's automagical enough.

Time — Jérémie Chassaing

Buried and scattered

File after file, project after project, finding traces of time in systems seems like a giant treasure hunt. Digging deep in the stack, shoveling heaps of code, turning classes over, time is nowhere to be found.

Hints take the form of asynchronous entry points, timers, threads and locks. Concurrent data access or synchronization primitives. Be it a date or a duration, meaning is shattered and understanding lost.

From a Domain-Driven Design practitioner perspective, it's obvious that there is something missing. A disturbing lack of model.

The essence of Time

Everyone knows what time is, but… giving a definition is not so easy.

A glimpse at Wikipedia[49] surely gives some clues:

> **Time** is the indefinite continued progress of existence and events that occur in apparently irreversible succession from the past through the present to the future. **Time** is a component quantity of various measurements used to sequence events, to compare the duration of events or the intervals between them, and to quantify rates of change of quantities in material reality or in the conscious experience. […]

but the following warning clearly states that there will be no single answer:

> **Time** has long been an important subject of study in religion, philosophy, and science, but defining it in a manner applicable to all fields without circularity has consistently eluded scholars. […]

This definition is referring to duration, that is actually referring to time…Maybe a more formal and physical definition could settle this.

The universal unit of time is the second. Since 1967 it is defined by the International System of Units[50] (SI) in this precise way:

[49] https://en.wikipedia.org/wiki/Time
[50] https://www.bipm.org/en/publications/si-brochure/second.html

> The second is the duration of 9 192 631 770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the caesium 133 atom.

The second is defined as a count of transitions. But those transitions are not Time. Transitions are just points in time.

There is something between those points, and this is precisely where time is. Time is the *nothingness* between those events.

Events

Let's step back a bit.

How do we feel time passing? By looking at a watch?

Then how to explain that an hour sometimes seems so long and sometimes passes in a flash?

Time seems slow and empty when we're bored while Time seems fast and full when we're busy with interesting things.

Boredom is not caused by the fact that nothing happens. Since there are actually always things happening. Clocks keep ticking, cesium atoms transitioning, the world spinning. But boredom still happens because these events seem of low interest to its subjects.

Events that don't affect the subject are filtered. Only events that change the subject substantially get noticed and are remembered.

The interesting or noteworthy things that happen act on personal state and change it. These are **meaningful events**. Things that happens and change people deeply.

Of course a lot of things happen between those meaningful events like moving and thinking. The blood flows through the body, but this is just a maintenance move. So it goes unnoticed. And maybe some things happen between state transition of a cesium atom, but since it's not possible to notice it and give it a meaning for now, it has no influence and just doesn't actually exist.

This perception of meaningful events is surely a reason why people say the time pass faster when old. In the 6 first years, any event is meaningful for kids. "Look dad! A white car! And here another one, and there another one!" Any event makes them change since they have no previous knowledge. Then has time goes by, people integrate knowledge and filter things they already know, they've already seen. When old, a year can more easily seem the same than the year before.

But some people continue to enjoy and learn as much as they can to still have a long now[51].

But when a meaningful event happens, the subject changes and is not the same before and after.

This is what time is about, and that's the reason it is one way.

Before >> Event >> After

Events define time by causality

[51] http://longnow.org/essays/big-here-long-now/

Time inside Systems

Tracking changes in systems is usually done by adding log and timestamps at strategic points. But it's seldom done initially. It's generally added later in the process to help understant when things go wrong.

Now, looking at the problem with a new understanding of Time it becomes clear that *Systems never change for no reason.* It's always because a meaningful event happened.

This event can be caused by a user interaction, a call from an external system, a sensor trigger… It can also be following by another event. But there is always a reason.

And when things change because it's midnight? It simply means that midnight is a meaningful event in the system. And this meaning has to be found. In financial markets, a lot of things happen a 4PM. But it's not due to the fact that it's 4PM. It's actually because the bell rang, and the market is closing. Noticing that the close bell rang is the reason, it makes it fare easier to handle the case when the bell rings earlier due to market exceptional situations, like market crash.

Where are those meaningful events in the System? Hidden in infrastructure code?

Greg Young would say:

> Make the implicit explicit!

Domain Events

There is no change in the domain that is not due to an Event. And Domain-Driven Design should definitely encompass them to create better models when time passing in the system is part of the problem.

Once starting to look for them, events appear everywhere in the Ubiquitous Language:

- When the client **has moved** to a new location, send him a welcome kit.
- When a **room has been overbooked** try to relocate the customer
- Every day at midnight, when **day changed**, evaluate last minute prices.

In all domains that are business related, Domain Events are relevant. They have to deal with time because business is mostly about time and money.

By introducing Domain Events in a domain model, Events and Time become explicit in the domain.

Time is now part of the Ubiquitous language and it becomes easier to provide a clear implementation for it.

Agents aka Domain objects on steroids – Einar Landre

From here to there and back again, Bilbo Baggins

Introduction

This essay explains dynamic domain modelling, why it is needed and the value from making it a first class citizen of Domain-Driven design. But before embarking on that journey, I would like to thank Eric for his seminal contribution to the software community. I will also thank him for all the great discussions we had battling Equinor's (Statoil) oil trading portfolio and writing papers for OOPSLA. That was a great experience.

Fifteen years has passed since the book was published. Back then, there was no iPhone, no Facebook, no Netflix and Amazon had been profitable for two years. Windows 2000 was Microsofts flagship operating system, Sun Microsystems was a leading tech company, Java was 9 years old and the relational database ruled the enterprise data centres.

Since then, cloud computing, big-data, mobile-apps, internet of things, edge computing, machine learning and artificial intelligence has become part of our professions vocabulary. New programming languages such as Swift, Scala and Go has entered the scene and old languages such as Python have resurrected and dominates data science.

It is easy to argue that our profession experience profound changes, changes that I think make Domain-Driven design even more important, changes that also require Domain-Driven design itself to change, adopting itself to the needs of a software defined world.

Domain complexity

The British system thinker Derek Hitchins argue that complexity is a function of variety, connectedness and disorder. We perceive things as more complex if there is greater variety among components, more connections between components and the connections are tangled instead of ordered.

The challenge is that we have two types of connectedness. We have stable connections that lead to structural complexity and we have arbitrary connections that leads to dynamic complexity.

Structural domain complexity is what we find in nested structures such as as the component hierarchis in products (airplanes, ships), retail assortments or project plans. The objects become complicated due to their internal state models, their rules and the depth of their connectedness and variability.

Dynamic domain complexity comes from the intercourses between autonomous components or objects. This is the complexities we find in dynamic systems. Objects might have high internal complexity but the dynamic complexity is created by their ever changing interactions and arbitrary connectedness.

Objects come and go, they might be lost to enemy action or communication loss. They might collaborate, compete, form teams and actions taken by one object might have direct impact on other objects available options.

Domain-Driven design addresses structural complexity. Entities, value objects, aggregates, repositories and services are structural building blocks that helps us to create order, reduce connectedness easing the management of variability within and between bounded contexts.

Dynamic complexity is not addressed at all. Vernon introduces domain events in his book and that is a good start, but we need more than events and the way events has been managed in context of enterprise messaging software.

Agents

The real world consist of dynamic systems. Dynamic domain complexity originates from asynchronous, concurrent, competing and collaborating processes.

Object oriented programming was created to study and analyse processes in context of a system through simulation and Simula provided the required support. The object oriented software community lost (by reasons unknown to me) its interest in dynamic systems and turned itself to programming languages. The study of dynamic systems was left for the control theory (cybernetics) and artificial intelligence (AI) communities.

It's worth mentioning that control theory and AI are siblings, both conceived at workshops in the late 1940ties. The differencing factor was that control theorists lent themselves to calculus and matrix algebra and the problems suited for that approach i.e. systems described by fixed set of continuous variables. The AI group did not accept those limitations and chose the tools of logical interference and computation, allowing them to consider problems such as language, vision and planning.

In the AI community the ideas of rational actions led to rational agents (agere Latin for to do) aka intelligent agents i.e. computer programs that does work. Of cause, all programs does work, but agents are expected to do more: operate autonomously, persist over time, adapt, change and create and pursue goals. A rational agent is an agent that acts to achieve the best possible outcome (Russel & Norvig).

An intelligent agent is a program that solves problems by observing an environment and execute actions toward that environment. Agents can act in roles, collaborate and interact with other agents including humans.

Software wise are intelligent agents objects that control their own execution thread, they are active, and they do interesting things. The problem though, very few see agents as domain objects and I think that has to change.

To illustrate that agents are domain objects I have provided a piece of Java code that outlines the anatomy of an agent:

```
1  public class MyAgent implements Runnable {
2
3      public void run() {
4          while true {
5            // Percept environment
6            // Select and execute action
7            // Update internal state
8          }
9      }
10 }
```

Intelligent agents is one of the cornerstones of artificial intelligence. They come in many flavours, spanning from Bots to space crafts and they are the key to understanding autonomy which is tied to the agents learning capability.

The hard part is the implementation of the agents reasoning method or agent function, mapping a given perception or goal with the best possible action. Agent functions can be very simple and very sophisticated, something I have tried to illustrate using a capability stack:

- Reasoning based behaviour implemented using cognitive architectures such as BDI and Soar.
- Continuous behaviour found in digital filters, mathematical control systems, fuzzy logic and neural networks.
- State driven behaviour, typically built from finite state machines.
- Simple behaviour is represented by memoryless functions such reading a measurement.

The most sophisticated agent functions implies use of cognitive architectures such as Soar and BDI in combination all other available tools in the toolbox. Soar was created in 1983 by John Laired and Allen Newell and is now maintained by Laird's research group at University of Michigan. BDI or more precisely Beliefs-Desires-Intentions was created in 1991 by Michael Bratman in his theory of human practical reasoning. There is ongoing research related to both Soar and BDI, much motivated by the defence sectors need for autonomous capabilities.

Soar and BDI are domain models of how the human brain reasons and transfer perception into action. Both architectures are supported by open-source and commercial implementations such as JACK, BDI4Jade, Gorite and SOAR.

Dynamic domain modelling

Late professor Kristen Nygaard used Cafe' Objecta as his system metaphor when teaching object oriented programming. When observing Cafe' Objecta we find objects who does interesting things

such as Waiter, Guest, Gatekeeper and Cashier and objects that define and describe things such as Menus, Food, Bills and Tables.

Behavioural modelling is about the objects that does interesting things. It starts with task environment and leads to the more detailed event and task model. The task environment defines the context and includes who the agent(s) are (Waiter), the agents performance measures (good dining experience), operational environment (the restaurant) actuators (voice, hands and feets) and sensors (eyes, ears).

The event and task model decomposes the higher level objectives into more detailed tasks using a repertoire of questions: What are the tasks to be performed? What event triggers the need for a certain task? What is the intended outcome from a task? What messages are sent and who are the receivers? Who performs the task? What events are created by executing tasks?

When developing an event and task model there is two vital factors to consider: firstly, what tasks will be performed concurrently and will they compete for the same resource? If yes, then we are heading toward race conditions and possible deadlocks and we need concurrent programming skills. Secondly, if there are tasks with time budgets i.e. tasks must be completed within given time frames we need real-time system skills.

To support the modelling of dynamic systems we need to add four concepts to our Domain-Driven Design toolbox:

- Tasks, the work to be performed by the agents.
- Agents, the objects who's perceives its environment and execute tasks.
- Agent function, how the agent maps its perceptions to tasks.
- Events, the things happens that triggers the need for a task to be performed.

By making these four first class citizens of Domain-Driven design we are well off and in position to build even richer and more capable domain models, the models needed to address Internet of Things, Industry 4.0, Artificial Intelligence and a world where software provided functions is ubiquitous.

For those who want to take it one step further Douglas, Russel & Norvig and Jarvis et al are all good reads. Hitchins for the special interested. An I expect that all of you already have read Eric or Vernon's books.

Reflections

Some might ask, what differs an agent from a micro-service? My answer is granularity. Agents are objects, they are in the end defined by the class constructor in the language of choice.

If your domain problem is best solved using a cognitive architecture, the advice is to look for an established framework. No reason to build this yourself.

Why now? - The Internet of Things and the move toward the software defined world changes the rules for business software. Back-office processes must be able to respond to events at the edge, and

they must be able to send new instructions to the edge process and devices in real time. - Automation of work begins with capturing the tasks.

I hope you now understand agents as first class citizens of many domains, and that they are domain objects on steroids.

References

- Douglas, Doing hard time, Developing real-time systems with UML, Objects, Frameworks and Patterns.
- Evans, Domain-Driven Design, Tackling the complexity at the heart of software.
- Hitchins, Advanced systems, thinking, engineering and management.
- Jarvis et al, Multiagent Systems and Applications: Volume 2: Development Using the GORITE BDI Framework.
- Russel, Norvig, Artificial intelligence, A modern approach, third edition.
- Vernon, Implementing Domain Drive Design.

Domain-Driven Design as Usability for Coders — Anita Kvamme

My Way into Modelling

Having a usability background, I learned years ago about a model which stated that if you are able to communicate the functionality by using a design model, which reflects the user´s mental model, you are more likely to get a system with a high degree of usability. This model is probably best known from Don Norman famous book The Design of Everyday Things published back in 1988.

A mental model is what the user believes he or she knows about the system at hand. The users´ mental model for a business solution will be influenced by both their mental model of the business and their mental model of how these types of solutions typically work.

The user unconsciously exploits this model to predict the system´s behaviour. To design a solution with high degree of usability, the designer model must be as close as possible to the user´s mental model. That way, a user will be more likely to correctly predict how the solution works.

As a UX person I love usability testing where users are thinking aloud when they perform their tasks, since this gives me a sneak peek into how they are thinking and by that getting real insights into their mental model. This insight can be exploited to adjust the designer´s model, and for me chasing the ultimate designer model, was my door to into the world of modelling.

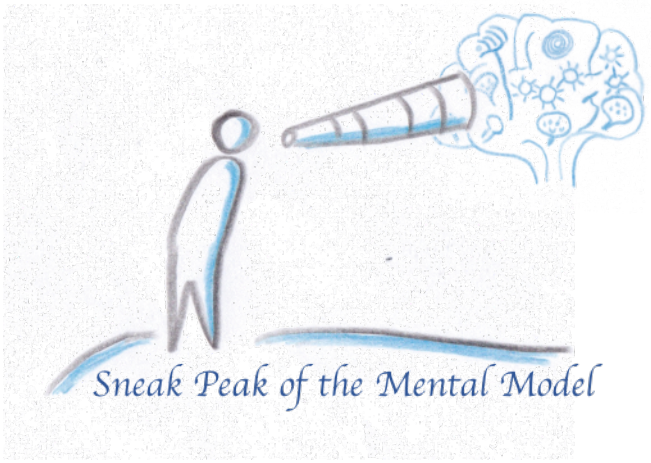


Figure 1.

Many years later the company I worked for had hired Eric Evans himself to help us using DDD. I clearly remember him on stage of our biggest auditorium asking if anyone had any stories about

modelling to share. There I was, having the stories based on years of experience with user interface design, but lacking the courage to share them.

What I wanted to share were stories from real projects where the focus on the designer´s model also lead to changes in how the technical solution was implemented. One time when I asked the rest of my team, “where does “concept X” belong in our design model”, we agreed that the concept would not make sense for a user and should not be visible through the user interface. After deciding that, it was a short way to remove the concept completely from the technical design, all the way down to the database model. In this case, the focus on usability influenced the codebase in a similar fashion as DDD want us to do, and this was years before the famous blue book.

The Coder´s Mental Model

Domain modelling is an important part of DDD, and this makes it easier to solve the complex business problems. However, I believe DDD has another benefit as well, namely making a good environment for coders to build a useful mental model. So, what do I mean by that?

Jessica Kerr said in her brilliant keynote at Explore DDD that a mental model must be made by continuous adding bits of understanding. She pointed out that this explains why there is so many Java Script frameworks out there. That is because it is much easier to make your own, and at the same time incrementally building the mental model of how it works, than to build a correct mental model of an existing framework.

With a complex business application having multiple bounded contexts, rewrite isn´t typically an option. So, we need another way of helping experienced coders and newbies in the team to build a shared understanding, and by that be more likely to build similar and useful mental models that will offer real guidance in the code jungle.

To apply the same pattern as used for years in the usability area, the goal should be to make a codebase structured in such a way that the coder´s mental model will help them to correctly predict how to navigate in, and enhance, the code. Because, when I find my way around the code written by others, it´s my mental model of the codebase that leads the way. And similar, when I code, it is my mental model of the codebase, in combination of my belief of how we as a team want it to be structured, that guides my coding.

The mental model is internal to each coder´s brain and will differ. As Jakob Nilsen, a well known usability expert, states that mental models are in flux exactly because they’re embedded in a brain rather than fixed in an external medium. Additional experience with the system can obviously change the model, but users might also update their mental models based on stimuli from elsewhere, such as talking to other users or even applying lessons from other systems.

However, even though we cannot predict the coder´s mental model of the codebase or decide that all coders should have the same model, it is possible to structure the codebase in such a way that it is more likely that the coder´s mental model will give correct predictions. So, what influence the coder´s mental model in this context?

- Inside the potentially huge amount of code which implements the domain behaviour, when applying DDD right by using the ubiquitous language explicit in the code, everybody that share an understanding of this language will have a common foundation that helps them to build a useful mental model. And coders not knowing the business domain will benefit from using time to learn the domain to enhance their mental model, instead of using time to wander around in a code landscape where all the words are more randomly chosen.
- In a huge and complex business area, having strict boundaries between different bounded context will make it possible for the users to predict what bounded context a given piece of functionality belongs to.
- Applying architecture patterns, as for instance the hexagonal architecture, will influence where to look for repository code, app service code and domain code. If the coder´s do not know this architecture pattern, acquiring this knowledge helps building this understanding.

Together all these aspects will highly influence the coder´s mental model of the codebase. However, learning more about the domain and the chosen technical architecture will only have a positive influence on the coder´s mental model given that the solution is implemented in a consistent manner. On the other hand, if the functionality is placed inside the wrong bounded context, if the domain logic has leaked into the repository, or if a change of the language isn´t reflected in the code, the coder´s mental model is likely to be misleading. Therefore, not following these DDD principles does not only make the IT solution building up technical debt, it also makes a weaker foundation for the coders to build a useful mental model.

Minimizing the Mental Model to Code Jungle Gap

The same way that people in the usability area for decades have worked against minimizing the gap between the designer´s model and the user´s mental model, DDD is a powerful set of principles to facilitate for a small as possible gap between the coder´s mental model of the codebase and the code. So for me, applying DDD isn´t just fun and useful when implementing complex business functionality, it is actually usability for the coders.

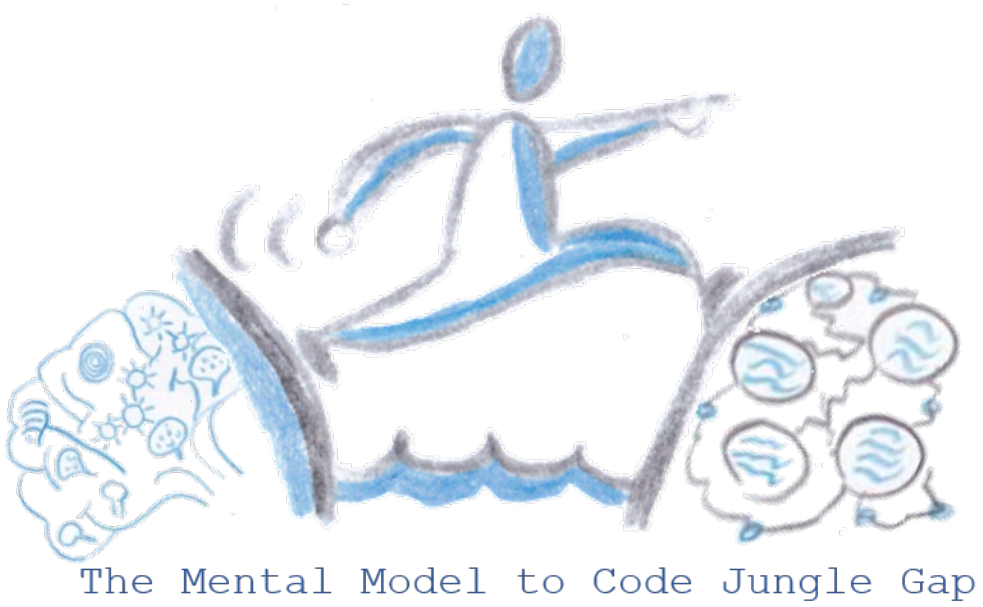


Figure 2.

Sometimes I wonder, if it is this usability aspect of DDD that is the explanation for when having started to apply DDD, it seems hard to stop. And who knows, maybe that is part of the reason why DDD is celebrating 15. years anniversary with increasing popularity.

References

- The Design of Everyday Things, author: Dan Norman, published in 1988.
- Jessica Kerr´s Keynote at Explore DDD 2018[52]
- Jakob Nielsen article about Mental Models[53]

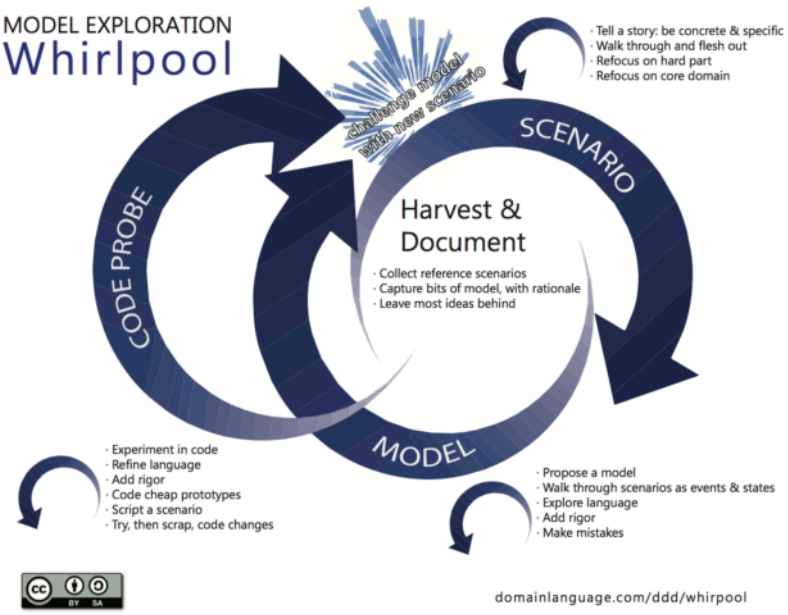
[52] https://www.youtube.com/watch?v=nVRUv30coyA
[53] https://www.nngroup.com/articles/mental-models/

Model Exploration Whirlpool – Kenny Baas-Schwegler

with EventStorming and Example Mapping

Introduction

People often ask for more concrete guidance on how to explore models, especially in an Agile or Lean setting. The model exploration whirlpool is Eric Evans attempt to capture such advice in writing. It is not a development process, but a process that fits in most development processes. The central theme revolving the process is to keep challenging the model. While the process itself for most is straightforward and easy to understand, there are not many concrete examples to find on how to do such a model exploration whirlpool. Most people when starting to use Domain-driven design (DDD) are looking for these practical examples. In this article, I will tell you my story of how I used the model exploration whirlpool by combining EventStorming, a technique that came from the DDD community, and Example Mapping, a technique from Behaviour Driven Development (BDD) community.



Harvest and document

We start model exploration with harvesting and documenting the current state. If you don't have a current state and going green field, don't worry we will get to that part. The primary goals are to collect reference scenarios, capture bits of the model and then leave most ideas behind. The starting point for this stage can be several of the following; A constraint formed out of a big picture EventStorming; A new project; or just a user story on the backlog. As long as there is a storyline to tell, it will be a good starting point.

One of my favourite tools to use is EventStorming, a flexible tool when doing collaborate discovery of complex business domain. In just a few hours you can harvest and document a lot of knowledge. It is essential however for the success of this stage to know who the right people are to invite. These are the people who know the domain, the domain experts. We want to be inclusive as possible, but there are exceptions. It is vital that we create a safe place in where knowledge can flow freely, a great facilitator can do miracles, but there are limits. People who are toxic are killing for a productive workshop, so we might want to decide to leave such a person out. But this person can have a lot of domain knowledge essential for this stage. It is perhaps wise at this stage to talk to this person before such a workshop and see if we can get some information out, or even better but harder, change this person's toxic behaviour so that this person can join the workshop.

The workshop

One important thing here is to find enough modelling space, preferably infinite space (if ever!). I look for rooms with at least an 8 meters long wall of where I can put a paper roll on, and we definitely need a whiteboard. Now we can start capturing the current state by doing a process EventStorm. We give the participant orange stickies and a marker. We ask them to write down domain events, a pattern later added in the reference book by Eric Evans. A domain event, in short, is something that is business relevant that has happened in the domain. For process EventStorming, we want to capture domain events of the current state and put them on the white paper in order of time. There will be a lot of chaos, so be sure to do some expectation management and explain that eventually, we will structure it.

Tell a story

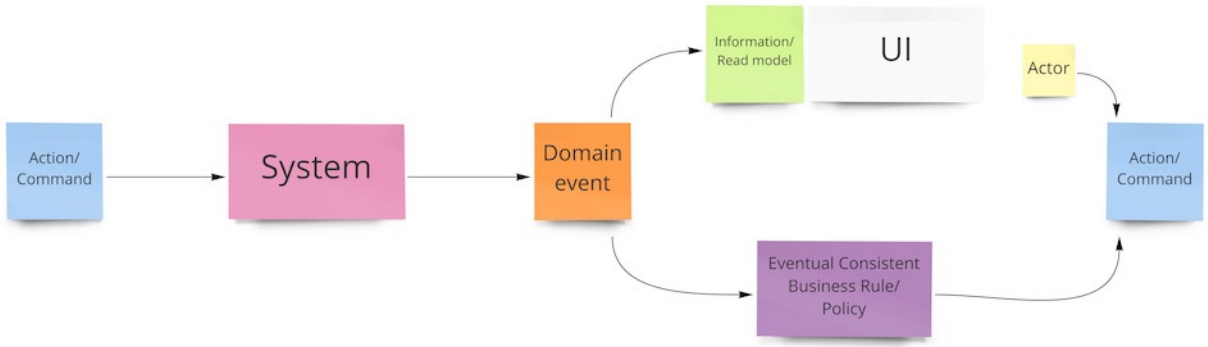
When everyone wrote down their domain events, it is time to tell a story. Let the attendees enforce the timeline now by making the story consistent. Remove duplicate events, but make sure they are in fact duplicates. Often people assume a Domain Event is the same even but isn't. Be concrete and specific. At this point, expect to find different opinions about the expertise. Mark these with a bright pink sticky called a hotspot, something that is noticeable from a distance.

Make the implicit, explicit

The core of visual meetings like EventStorming is that we discuss only explicit and visible things. There is just so much knowledge we can capture in domain events and hotspots; we can capture

more types of knowledge with other colours. The standard EventStorming colours are (but make your legend with the colours you have available):

- Blue: Action/Command
- Long Pink: (external) System
- Long Lilac: Policy/Eventual Business Constraint
- Green: Information



For more information about eventstorming read the book by Alberto Brandolini[54]

The basic flow will look like this, but the critical point here is to make communication explicit, if it is explicit for everyone in the room, that is enough. If we don't know, try and follow this flow. Remember to make the implicit, explicit. Discussions that do not take place on the paper roll needs to be made explicit. Sometimes it is hard to make it explicit in just a few stickies, that is why we need to have a whiteboard at hand where we can make sketches, drawings or write down bits and pieces of the model.

Bias blind spot

Now that we think the story is complete we want to bring in Example Mapping. Most of the time people will now believe that bringing in another tool is waste. We got our story visualised right; we got the full story? The problem is that everyone is subject to cognitive bias, especially when we get information overload. We notice things already primed in memory or repeated often; this is called the context effect and attentional bias. We are also drawn to details that confirm our own existing beliefs; this is called the observer effect and the confirmation bias. Especially the bias blind spot, noticing flaws in others is more easily than yourself is dangerous during our exploration of the domain. To battle these biasses we need to use different viewpoints, other tools.

Example mapping helps us here because it focusses more on specific examples. Make room on a different part of the model space, either next to your EventStorm or on a separate paper roll. Start

[54] https://leanpub.com/introducing_eventstorming

with the storming with writing down examples on (usually green) stickies in the form of friends episodes. Friends episodes always begin with the one where. Think out of the box, and see where these examples affect the current EventStorm. Is it a gap in the current system, or is it a gap in knowledge which is futile to go on, mark it with hotspots, make it explicit!

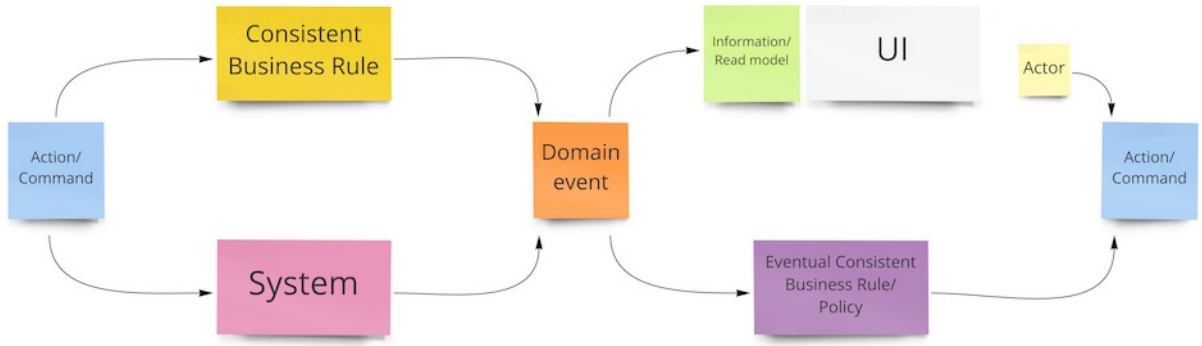
Now at this point, with the people in the room, we probably harvest enough insight of the current state. Now we can already start seeing bounded context emerge, or see where systems began to entangle with each other, and the boundaries are not made explicit. This first part of the workshop will take about two to three hours, a minor investment to the knowledge that is gain. When we did so much knowledge crunching of the as-is situation, it might be wise to stop the workshop here and sleep over and process the acquired knowledge.

Scenario

Now that we acquired all that as-is knowledge of the system we want to do the same for the to-be situation. For the to-be, we create a new modelling space for us to do EventStorming again, use a new paper roll, and start storming domain events for the to-be situation.

Walkthrough and refocus on the hard part

As soon as we have all the events stormed we want to do the same as last time, walkthrough events, enforce the timeline and remove duplicate domain events. What we now want to do it refocus on the hard part, to find where the complexity is. We do this by introducing a new concept. Instead of the long pink sticky, we can also use the long yellow sticky for consistent business rules. Writing each rule per sticky will help us, later on, refactor them more efficiently instead of writing several down on one sticky. A consistent business rule will always be in front of a domain event.

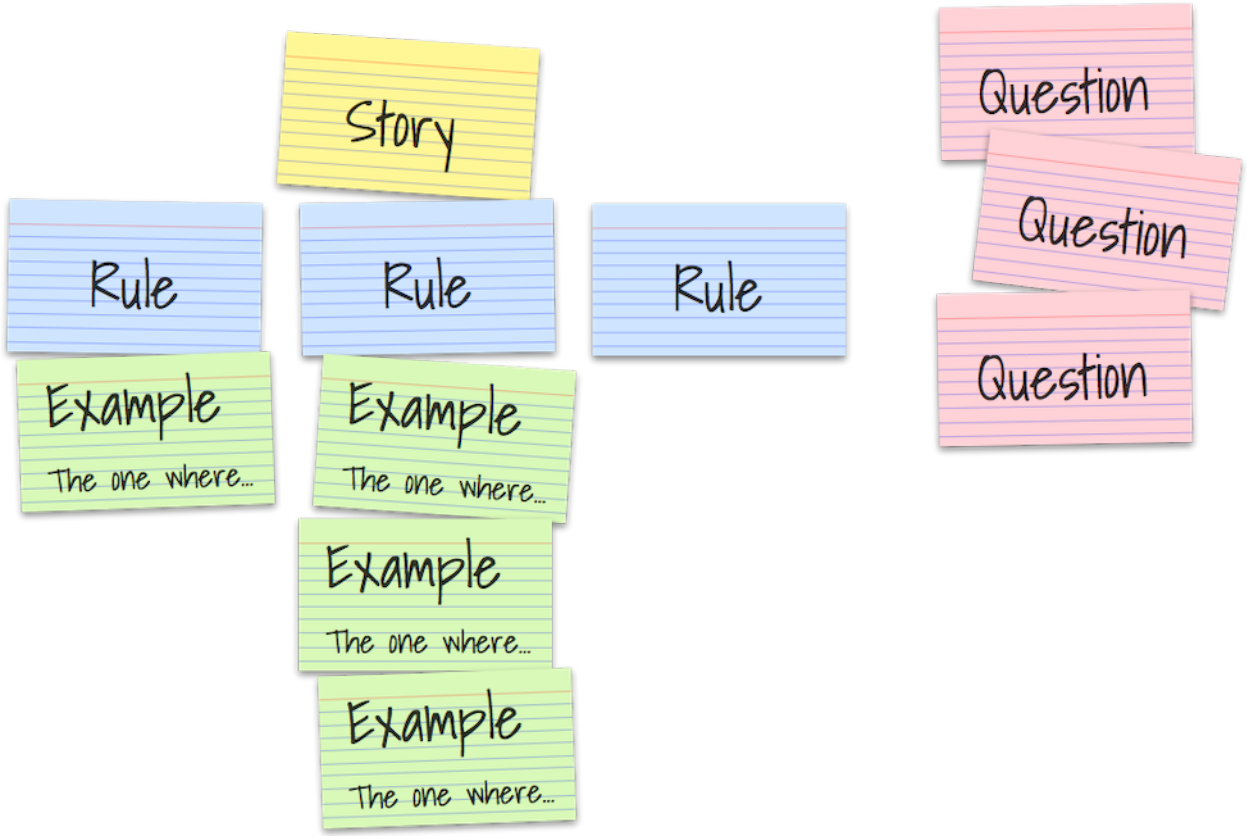


We want to look for the consistent and eventual consistent business rules (the yellow and the purple) first, and focus on these part. Make the story consistent by adding in the other coloured stickies. Focus on the language used; it is essential to find and see where words become ambiguous. Also,

start adding in actors to make more visible who is responsible for what part of the story. All this information combines is what defining a proper bounded context is all about.

example map

Like with the as-is, we will use example mapping again. First start with the storming part again, writing down examples on a green sticky either next to the EventStorm or a separate wall on a different paper roll. Only this time we will also go further in the example mapping, by structuring the examples in vertical rows by a business rule. Write business rules on a blue sticky above the vertical row of examples. Important is to only have one business rule per vertical row. Having only one business rule means that specific examples can happen multiple times, but will focus more on a different business rule.



Â© cucumber.io[55]

The business rules will match the business rules on your EventStorm. You will most likely also find new business rules you need to make explicit. When this happens, you might need to adjust your EventStorm with our newly acquired information. The goal of both tools is to share knowledge and explore complex business domains, so be careful not to go all out on making the two consistent.

[55] https://cucumber.io/blog/2015/12/08/example-mapping-introduction

Model, slice, formalise and code probe

With our newly required knowledge, it is now time to start modelling. We first will explore different models and see how the models will hold up against the EventStorm and the examples on your example mapping. Try and find at least three models and quickly iterate over them. Once you end up with a workable model, we can now slice our example map. Discuss which business rules are the most important and start and formalise the examples.

With our workable model and formalised examples, we can now start coding. Because we know what the system needs to do based on our formalised examples it is easy to use Test-Driven development to cheapy write a prototype of our model in code. This way we continuously refine our language and challenge our model.

Domain-Driven Design as a Centered Set — Paul Rayner

This essay has been adapted from Paul's blog post "BDD is a Centered Community," originally published in 2015.

Dan North mentioned in the CukeUp 2015 panel the notion of a community being a "bounded set," and I pointed out that the same theology also talks about the notion of a community being a "centered set." The original context of the panel discussion concerned BDD (behaviour-driven development), but I believe the idea applies equally to Domain-Driven Design (DDD). DDD is a centered set, rather than a bounded set. To extend this further to the community that has grown around DDD over the last 15 years, we might say *DDD is a centered community, rather than a bounded community.*

Let's be clear what is meant by bounded vs. centered set as it applies to a community, and how that might apply to DDD. These terms originated in a theological context with the writings of anthropologist and missiologist Paul Hiebert, in terms of what makes a person a Christian. I'm going to quote and adapt a significant amount of the content from his original article in what follows to make it relevant for those without a background or interest in theology.

Bounded sets

In his 1978 article, "Conversion, Culture and Cognitive Categories", Paul Hiebert notes that many of our words refer to bounded sets: "apples," "oranges," "pencils," and "pens," for instance. What is a bounded set? How does our mind form it? In creating a bounded set our mind puts together things that share some common characteristics. "Apples," for example, are objects that are "the firm fleshy somewhat round fruit of a Rosaceous tree. They are usually red, yellow or green and are eaten raw or cooked."

Bounded sets have certain structural characteristics — that is, they force us to look at things in a certain way. Let us use the category "apples" to illustrate some of these:

a. *The category is created by listing the essential characteristics that an object must have to be within the set.* For example, an apple is (1) a kind of "fruit" that is (2) firm, (3) fleshy, (4) somewhat round, and so on. Any fruit that meets these requirements (assuming we have an adequate definition) is an "apple."
b. *The category is defined by a clear boundary.* A fruit is either an apple or it is not. It cannot be 70% apple and 30% pear. Most of the effort in defining the category is spent on defining and maintaining the boundary. In other words, not only must we say what an "apple" is, we must also clearly differentiate it from "oranges," "pears," and other similar objects that are *not* "apples."

c. *Objects within a bounded set are uniform in their essential characteristics.* All apples are 100% apple. One is not more apple than another. Either a fruit is an apple or it is not. There may be different sizes, shapes, and varieties, but they are all the same in that they are all apples. There is no variation implicit within the structuring of the category.
d. *Bounded sets are static sets.* If a fruit is an apple, it remains an apple whether it is green, ripe, or rotten. The only change occurs when an apple ceases to be an apple (e.g., being eaten), or when something like an orange is turned into an apple (something we cannot do). The big question, therefore, is whether an object is inside or outside the category. Once it is within, there can be no change in its categorical status.

What if DDD was a Bounded Set?

What happens to our concept of "DDD" if we define it in terms of a bounded set? If we use the above characteristics of a bounded set we probably come up with something like the following:

a. *We would define "DDD" in terms of a set of essential or definitive characteristics.* These characteristics, such as applying building block patterns, strategic design practices such as context mapping, or technical approaches such as event sourcing, would be non-negotiable for people doing DDD.
b. *We would make a clear distinction between what is "DDD" and what is not.* There would be no place in between. Moreover, maintaining this boundary would be critical to the maintenance of the category of DDD. Therefore it would be essential to determine who is doing DDD and who is not, and to keep the two sharply differentiated. We would want to make sure to include those who are truly doing DDD and to exclude from the community those who claim to be but are not. If DDD was to be a bounded set, to have an unclear boundary would be to undermine the very concept of DDD itself.
c. *We would view all "DDD practitioners" as essentially the same.* There would be experienced DDD practitioners and beginners, but all are doing DDD, since they are within the boundary. Homogeneity within the community in terms of belief and practices would be the norm and the goal.

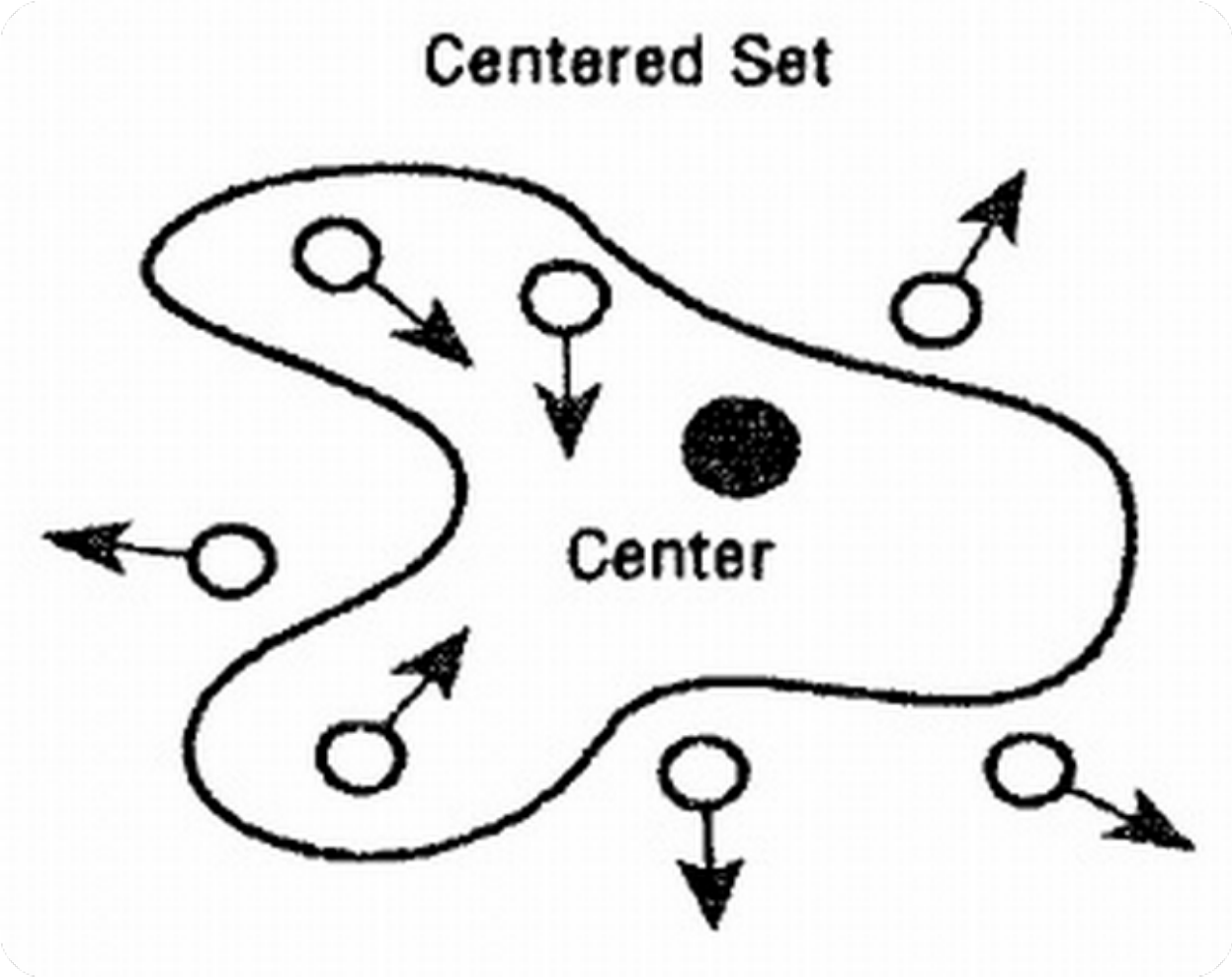
If we think of DDD as a bounded set, we must decide what are the definitive characteristics that set a DDD practitioner apart from a non-DDD practitioner. We might do so in terms of belief in certain essential "truths" about DDD, or strict adherence to certain essential DDD "practices."

However, DDD as understood by leaders within the community is clearly NOT a bounded set. Rather, it is a centered set. Let's see what we mean by that.

Centered Sets

There are other ways to form mental categories. Hiebert says a second way is to form centered sets. A centered set has the following characteristics:

a. *It is created by defining a center, and the relationship of things to that center.* Some things may be far from the center, but they are moving towards the center, therefore, they are part of the centered set. On the other hand, some objects may be near the center but are moving away from it, so they are not a part of the set. The set is made up of all objects moving towards the center.
b. *While the centered set does not place the primary focus on the boundary, there is a clear division between things moving in and those moving out.* An object either belongs to a set or it does not. The set focuses upon the center and the boundary emerges when the center and the movement of the objects has been defined. There is no great need to maintain the boundary in order to maintain the set. The boundary is not the focus so long as the center is clear.
c. *Centered sets reflect variation within a category.* While there is a clear distinction between things moving in and those moving out, the objects within the set are not categorically uniform. Some may be near the center and others far from it, even though all are moving towards the center. Each object must be considered individually. It is not reduced to a single common uniformity within the category.
d. *Centered sets are dynamic sets.* Two types of movements are essential parts of their structure. First, it is possible to change direction — to turn from moving away to moving towards the center, from being outside to being inside the set. Second, because all objects are seen in constant motion, they are moving, fast or slowly, towards or away from the center. Something is always happening to an object. It is never static.



Centered Set

Illustrations of centered sets are harder to come by in English, since English tends to see the world largely in terms of bounded sets. One example is a magnetic field in which particles are in motion. Electrons are those particles which are drawn towards the positive magnetic pole, and protons are those attracted by the negative pole. The diagram here is another way of visualizing a centered set

DDD as a Centered Set

The core DDD principles have been enumerated quite clearly by Eric Evans on a number of occasions. See, for example, his Explore DDD 2018 conference keynote "DDD Isn't Done: A Skeptical, Optimistic, Pragmatic Look". DDD has a broad and diverse distributed community with principles at the center that endeavors to promote a growing body of knowledge conjoined with an emerging and changing cloud of practices. Focusing on boundaries and exclusion is not what the DDD community has ever been about.

There is little interest in the DDD community in adopting a bounded set approach, and staying static

by "locking down" the body of knowledge or cloud of practices that currently makes up DDD. Rather, the DDD community needs to continue challenging the status quo and incorporating new learning and practices revealed in the future. And, as Eric pointed out, even the core principles themselves should not be above challenge, or applied blindly.

In contrast to a bounded set, how might DDD be defined as a centered set?

a. *A DDD practitioner is be defined in terms of the center — in terms of the principles, values and goals that the DDD community holds to be central.* From the nature of the centered set, it should be clear that it is possible that there are those near the center who know a great deal about DDD, but who are moving away from the center. On the other hand there are those who are at a distance — who know little about DDD because they are just starting to learn it — but they should still be considered to be DDD practitioners.
b. *There is a clear division between being doing DDD and not doing DDD.* A boundary is present: To pick an extreme example, a team doing waterfall (serial lifecycle phase gate) development with no collaboration between domain experts and developers, and not doing any model exploration could not be said to be doing DDD.

But with a centered set there is less stress on maintaining the boundary in order to preserve the existence and purity of the category than with a bounded set. There is also no need to play boundary games and institutionally exclude those who are not truly part of the DDD community. Rather, the focus is on the center and of pointing people to that center. Inclusion, rather than exclusion, is the name of the DDD game. At least, for a centered set that should be the goal. c. *There is a recognition and encouragement of variation among the DDD community.* Some are closer to DDD principles in their knowledge and practice, others have only a little knowledge and need to grow. But - whether novice or expert or somewhere in between - all are doing DDD, and are called to continuously seek to improve and grow in their understanding and practice of delivering value early and often.

Being a centered set, growth thus is an essential part of practicing DDD. When a team begins doing DDD, they begin a journey and should strive to continue to move towards the center. There is no static state. Learning DDD is not the end, it is the beginning. It is not the end, it is the means to a greater end: at a minimum, creating better software to provide better outcomes for customers and stakeholders.

We need good quality DDD education, mentoring and coaching to teach DDD to beginners who join the community in the years to come. Perhaps more importantly, we must also think about the need to continuously improve and inspire novices and practitioners alike to move beyond the human tendency to follow recipes and so-called "best practices" and experiment with tailoring design principles and practices to their own unique context. Also, I contend that for a healthy centered set, quality constructive disagreement should be embraced and encouraged. Dissenting ideas should be understood and evaluated to see if they have merit, rather than succumbing to the unfortunate tendency to become an echo-chamber.

I submit that the agile community in general should also be considered a centered set, with the agile manifesto as the central value statement for the movement. Whether DDD, or agile in general, being

a centered community means seeking not only to uphold but also increase the gravitational pull of the principles and values at the center.

References

BDD is a Centered Community Rather than a Bounded Community[56]

Paul Hiebert, "Conversion, Culture and Cognitive Categories." In: Gospel in Context 1:4 (October, 1978), 24-29.

Centered set diagram sourced from academia.edu[57].

[56] http://thepaulrayner.com/bdd-is-a-centered-rather-than-a-bounded-community

[57] http://www.academia.edu/6810466/Understanding_Christian_Identity_in_Terms_of_Bounded_and_Centered_Set_Theory_in_the_Writings_of_Paul_G._Hiebert

DDD — The Misunderstood Mantra — James O. Coplien

Eric, it is a great honor and pleasure that I would find myself in the company of those invited to address you on the 15th anniversary of your *magnum opus*. I too, have been with this process for 15 years, and have watched as both of our ideas have grown. Meeting you at *DDD Europe* was a highlight for me and was my first opportunity to learn from you first-hand. I still yearn for us to continue that process, and hope we will meet face-to-face again soon, In the meantime, this is like a letter to you that primes the pump for those discussions. I offer these ideas with the greatest respect and great expectations of what we'll learn together next.

This is a great opportunity to revisit the topics of some of our past discussions, to reflect on our shared aspirations, and to kindle the fire for the exchange of the ideas we carry in our hearts. Many of the thoughts relate to how to present the message in a way that draws people beyond superficial understanding (and misunderstanding) into the depth of ideas you were trying to convey, as we have discussed. It is fun and powerful to go forward on this little project as a surprise for someone who I count not only as a seasoned and thoughtful colleague, but also as a friend. And it is still my great hope that I will some day join you there across the pond to carry on this conversation in person.

Foreword

When I spoke at DDD Europe in 2016, Eric Evans and I commiserated that few people really understand Domain-Driven Design (DDD). It fits the old joke about teenage sex: everybody is talking about it and wants to do it, very few are really doing it, and those that are doing it badly. Even *I* find myself scratching my head about it, and in talking to Eric, even *he* would write the book differently today than when Addison-Wesley asked me to review it in its nascent days.

These reflections mark one perspective on fifteen years of the history of these ideas. Maybe fifteen years is too early to assess a work's place in history; that will take another ten years, at least, I think. But perhaps my current reflections can both give Eric pride in what he has accomplished and challenge him, along with his colleagues, to add new thoughts and insights to the existing base. Much of what I say here, I have no doubt that Eric already has on his mind. If so, so much the better.

First Impressions

Back in 2002, one of the first things that struck me about Eric's book was that he wasn't just another fundamentalist of object-oriented programming. A domain view, particularly in design, was difficult

to find in the literature of the time. The industry was, as ever, struggling to remedy its shortcomings. My own work had found domain-level thinking to be priceless in lifiting us out of object myopia, while most of the industry was stuck in the "just do it" mantra that one so often found in the object community since it became fashionable. Patterns had made some inroads to temper this perspective but there very few other ideas that had gained mindshare at this level. Eric's work was opening some long unopened treasure chests.

The most important contribution that Eric has made, in my opinion, is to raise awareness of what he calls the *ubiquitous language*. Though it is not likely ubiquitous and is certainly not a language, such a collection of words grounds the dialectic from which great design emerges. More on that later.

But two things bothered me a bit about the book. One was it didn't evoke any of the massive domain analysis literature. In the canon of computer science domains delineate areas of focus or knowledge both in the application and solution sectors of the business. It takes good analysis to find domains that are resilient under program evolution, and the book wasn't attientive to that — though the area is well-trodden. Maybe I invested a bit of my ego in this perspective because of my own work in this area on *Multi-Paradigm Design* from four years earlier, which is a discipline way of doing what one might well call Domain-Driven Design. No one else was citing that literature, either: much of it tied into the Software Engineering community, and it would be to put it mildly to say that there was little connection between software engineering and the object community at that time. Yet it puzzled me that someone using the all-important word *domain* in their book title wouldn't have paid homage to the rich publication legacy of *domain engineering* and *domain analysis*. (Google gives 183 million hits for *domain engineering* and 116 million for *Domain-Driven Design*.)

More broadly, the book seemed to evoke many of the same sentiments as one could find in XP, which was about five years old at the time. So the book was a paradox that both distanced itself from the rhetoric of the time, but at the same time embraced that era's downplaying of analysis. XP had consciously set off 180° out of phase with conventional wisdom. To do good *analysis* was part of the conventional wisdom of the time: to engage end users to better appreciate the business perspective before jumping into design. Instead, XP would use an on-site customer as an insurance policy against the possibility of any gap between user needs and developer fantasies. DDD found itself grounded in the nerd *zeitgeist* of the time. It would embrace design while consciously distancing itself from analysis.

There is no denying that the analysis culture of that era was badly in need of a reboot. Broad practice still evidenced widespread practices we had associated with "waterfall development," though even by this time the term *waterfall* had become a four-letter word with which few people would identify. The world was starting to run *from* waterfall. It was an uncomfortable era in which people had not yet substantiated a cause *to* which they should run. Frameworks, patterns, and programming language features had expanded to fill the mindshare space of the time. There were certainly those who were waiting for the next method, and in particular a method that would scale beyond XP — which made no claims for being able to support anything beyond single-team developments, in the face of management realms from the previous era who still believed in scaling. (And, sadly, this is a perspective that has resurfaced in the methods world here some 15 years later.)

The new *zeitgeist* had thrown out the proverbial baby with the bathwater. Much later research would in fact point out that the emperor's new extremes were as bad as those of the previous regime, as we would find in the landmark papers of Martin, Biddle, Noble, Abrahamsson, Siniaalto, and others. Martin found on-site customer to be "unsustainable" (*The XP Customer Role*). Many had found that TDD did not live up to its promises, including Abrahamsson and Siniaalto. But back in 2002, in the shadow of an Agile Manifesto that was scarcely a year old, it was almost Communist to say that you weren't agile, and you would certainly incur the wrath of the mainstream method talking heads of the era of your notion of "agile" did not include On-Site Customer and TDD, or if it whispered the term "analysis."

Domain-Driven Design was properly aligned with this perspective, but the framing was decidedly un-XP-like. The book had all the trappings of a mature, well-thought-out and practiced approach with its own disciplines, artefacts and, notably enough, its own vocabulary. All of these gave the book a well-deserved air of authoritativeness. I think that what the agile folks had discarded in the analysis area, DDD tried to push back into design. We have learned over the years that it is folly to separate design, implementation and testing of complex software, and that was one of the first trappings of waterfall to go out the window. But the analysis / production dichotomy remained: for example, in Scrum, this dichotomy is personified in the Product Owner role. Here, in the DDD book, it seemed like there was an attempt to bring analysis concerns to the design table. And of course this is useful in theory, or in the hands of highly experienced and disciplined developers. Practice begs greater moderation and deserves to celebrate analysis more than the DDD book does but much less than one found in the prevailing practice of the era. As such, the DDD book was the product of a stormy, ongoing dialog. Eric decided to put a stake in the ground where he did, based on his experience. I can only think that his experience was uncharacteristically graced with developers and leadership who had keen insight or business knowledge, or both. In that case, *anything* works, and the DDD world model was as good a way as any to structure the work.

I think it bears emphasizing that Eric was doing anything here but going with the flow. It's important for me to say that because, as someone who was skeptical of many of the pseudo-agile practices the late 20th century, it would be easy for me to lump Eric together with the rest of the opportunists on the basis of the overlap between his ideas and theirs. What he was proposing was radical to the agile folks in its degree of structure, and radical to the traditionalists in eschewing analysis. That he included disciplines, structures and vocabulary who by their very nature flew in the face of the fashion of the time, says that Eric was motivated by something deeper than selling a book or seeking fame. I wouldn't get to know him personally until 2016: It was immediately obvious to me, and I think it is obvious to anyone who knows Eric, that the last thing he is seeking is glory. He is motivated out the feeling of having discovered something deep and right, and maybe something that not many people have discovered on their own. I may disagree with him on several points of his thesis, but it somehow doesn't matter. We share the same process of inquiry and the same motivations. That's enough for me. And it lays a foundation for the analysis and counterpoints here. But I do digress.

I have often thought of the DDD book from the perspective of the domain of complex system development. My career has led me to study that domain as a phenomenon or object in its own right for the past 35 years. Over this time we have started to develop a theory, if one can call it that, of complex system development, and its theory borrows heavily on those outside our field such as

Christopher Alexander, Peter Senge, and Russell Ackhoff. Scrum is one of the first modern methods to explicitly draw on that theory. But the DDD book was hit and miss about drawing on any theory that would transcend development culture and perhaps compensate for accidental differences in personal insight and experience. And, as any such thing drawn from one's own experience, it works some of the time and it doesn't work some of the time. Though different from XP it faced the same challenges of applicability.

However, there is a weightier consideration here that has nothing to do with Eric or with DDD per se, but which blankets our entire discipline and which should be reason for concern. As one example, consider domain-specific languages (*DSLs*, also known as *application-oriented languages* or *AOLs*) which were in broad use and were well-defined by the early 1990s. Martin Fowler had taken the time-honored notion of domain-specific languages and started using it in the sense of conventions embedded in a program of an existing general-purpose language. There is a kernel of insight behind this to the degree that well-chosen identifiers in a nicely structured language make the code easier to understand. Still, it's about more than the identifiers, as language is about how we structure the relationships between the words. Even most modern programming languages trace their syntax back to FORTRAN (declaration, assignment) and the basic types and semantics of the popular languages are still the algebraic operators. While computing was originally about playing with numbers, we've come a long way, and true DSLs go beyond a mere vocabulary to a true language.

Languages have not only syntax, but semantics — and some of the most powerful semantics of a language lie in its idiomatic constructs. And most fascination with what has come to be called DSLs is at the level of vocabulary rather than language, and the concern at hand rather than the domain. So contemporary DSLs have become a thing of straining out the gnats of local expression while ignoring the camels of overall system structure. A language encodes a way of thinking about form, and much of the discipline of domain-oriented thinking is about the forms around which they organize. Expressing those forms in a suitable vocabulary is almost a footnote. The early 1980s tradition of AOLs and DSLs built languages from scratch to capture these fundamental forms, while the emperor's new DSLs use the same languages (grammar, syntax, and base semantics) while just giving you a new dictionary. It is a haunting parallel to the pretense that a ubiquitous language is either a language or ubiquitous.

Yet if you talk to a post-2000 new age nerd, they will refer back to the Fowler notion of DSLs if they know them at all. Perhaps the reason is that to do a great DSL requires an excruciating analysis investment: with analysis out of fashion, it became necessary to exapt the concept and create something else. Or it may be because DSLs tend to do very badly in domains that aren't rock-solid stable. The domain of mathematics is pretty stable so DSLs like Excel® do pretty well; parser theory is sound and stable so *yacc* and *bison* don't change that much. (The frequent *antlr* changes are annoying and seem to amount to syntactic saccharin and catching up with status quo.) Yet the emperor's new DSLs aspire to be the heart of a DDD approach: they are design rather than analysis; they build on somewhat extensible languages to embrace the concept of a ubiquitous language; and unlike rock-solid languages like Excel® or *yacc*, any coder can evolve the language at their convenience.

I think Eric tried to show us a North Star called a *domain* that should serve as the oracle for our

design decisions. One shouldn't casually change a DSL without consulting a domain. I feel that Eric's domains, for whatever reason, are more conventions than laws, and I have always liked the traditional domain analysis community (think Neighbors, Lai, Weiss, Parnas, and others) for being close to what they claim are inviolate foundations of the businesses their software served. But Eric's domains are certainly more axiomatic than a developer's daily sense of a term's importance, of how much they need to type to express a concept. and of the new age DSLs that seem little more than macros on steroids. And if his notion of domain missed the mark, his notion of ubiquitous language hits the bullseye — even if it is only about a vocabulary. Words mean things, and I think few people understand that like Eric does.

Reflections

Guild not the lily

I spoke above of my frustation in not being able to find a theoretic grounding for DDD ideas. It brings to mind a radical proposition: Perhaps, in fact, there is no theoretical grounding to program structure, any more than there is a formal grounding for structuring a building. Sure, a building must stand, and a program must compile and link, but those are more engineering concerns that concerns of science. Even from an empirical perspective it is difficult to tease out theories and principles of good design. And the book *does* underscore those that we agree about, such as separation of concerns and attentiveness to APIs.

For example, the track record of the domain analysts is mixed. We have successes like *yacc* and *Excel* whose semantics are grounded in axioms of computation. One can make similar arguments for *SQL*. Most complex domains of software development do not enjoy such formal groundings, sot it becomes devilishly difficult to put analysis stakes in the ground. I have seen many of these efforts, rooted in domain analysis, flounder, founder, and ultimately fail. They tend to recall the heavyweight, overly front-loaded efforts of the waterfall 1970s.

This raises a question about whether there is a separate, viable view of programming-as-craft that breaks with software engineering tradition. I for one hold many software engineering platitudes in minor contempt, but perhaps I should join Eric and encourage colleagues to take this disdain even further. Eric's book is a strange book in that, at first glance, it treats topics in the neighborhood of software engineering — most notably, those tried-and-true universals I mentioned above. Both the software engineering literature and Eric's book lack credible grounding for their core ideas, but that doesn't mean the ideas are wrong. And it's unlikely in either case that we will find universals.

In some way I may have fallen into the trap in the same way that we criticize the fundamentalists of object-orientation, top-down design, or waterfall. We have precious little evidence that any of them "work" or not. Waterfall put us on the moon. For example, I can view DDD as what formally is called a *single hierarchy system*. DDD seems to blur coding-time and run-time concepts (a good thing), but there is one perspective and philosophy that stands behind a single structure of bounded contexts. The problem is that they are bounded. Real boundaries rarely exist in the real world. The

accounting department and the UX designers don't communicate through well-defined memos and protocols, or even through documents or well-structured meetings, but may exchange the most crucial market needs around the coffee machine, If there protocol is not well-delineated, the protocol of their software artifacts should be equally porous.

There is light from another land here. I have seen many products thrive in mature domains such as telecom and finance by building on what even I would call a good domain analysis. But rather than relying on any method to distill the domain knowledge into design, these products delved into the patterns both of the business and of the domains of past practice. A seasoned telecom developer knows it is folly for a system to feature a *call* object (the system may use *half calls* instead — a key domain knowledge insight) — a good example of where domain knowledge trumps a naive object method. And Eric's work has a glimpse of that light to the degree that DDD evokes timeless structures of software construction. There's nothing wrong with that. As with many of Eric's ideas, the problem may be in that programmers supplant the critical knowledge of business domain patterns with Eric's patterns. You won't find *half call* in *Domain-Driven Design*. What one does find in the book is useful in the sense that facility with the technique of laying one brick upon another is useful. Most readers, seeing these ideas cast as patterns, extrapolate their substantiation as adequate to build a cathedral.

In summary, gilding the lily fails in two ways. Eric would be the first to agree that we should not puff up a tool or method to a stature grander than that which we are building; yet, I find many DDD shops using either DDD vocabulary or programmer vocabulary instead of the vocabulary of the business they serve (how ironic). On the other hand, we should not pretend that even the most mature domain is without complexity to the point of being able to be codified, in the way that the analysis folks of old would have us believe possible. *Ne quid nimis*.

Bring Back Analysis

With the rise of agile I have unfortunately seen the rise of a disgusting form of hacking I call "green bar fever." It's based on a rapid-feedback Pavlovian cycle where people make changes to their code that are as minimal as possible, re-run the test, and pray for the Green Bar. They tweak the code more than the test — as though the test for some reason was written when they had taken the smart pills while the code was written when they had not — and often end adding to technical debt.

As computers have become faster and as almost all programmers have at least one high-powered machine sitting on their lap, we have sought increasing opportunities for automation. Great design thinkers from Taichi Ōno to Elon Musk have warned against the evils of too much automation, and there are plenty of data and models to support the fact that automation lowers quality and may increase rework. As Toyota removes robots from their assembly lines software engineers are automating more and more work. The tools and technologies of the new millennium are starting to take on the prestige that technology held in the 1950s and 1960s, when we all envisioned robots cooking our meals for us within a few years,

DDD doesn't fall into this trap in an obvious way, but it has come to be understood as being in much the same space. It's about *design* rather than *analysis*. I am currently building a new house. Picking

a site takes years. Laying out the house on the site takes weeks or months. Choosing a floor plan and kibitzing with the architect takes weeks. Selecting the appliances and materials takes quite a while. The house itself — a sturdy, Norwegian-designed Trelleborg log house — goes up in a couple of days. Software people instead completely throw away analysis (DDD explicitly has no room for the activity of analysis) and go right into design.

The belief, I think, is that software is soft enough and languages expressive enough that we can iterate our thought process in the implementation. Experience after experience, and the mere fact that we have the phrase "technical debt" in our vocabulary stand testimony to the folly of this perspective. There *are* viable "tools" to help those who want to be concrete: prototyping, set-based design, modeling, formal analysis, and a host of others. These things take nerds out of their comfort zone of Java, Python or Ruby. So they stay in the code. And they have crafted an image of DDD that cheers them on.

Programming languages and their design methods bring little insight into end-user needs, except in the case of domain-specific languages in the broad sense, rather that the narrow sense they have taken on as described above. In a new exposition of DDD into the wild, it woiuld be good to acknowledge the place of sound analysis, or at least domain expertise, as a companion to the good design practices that are already there.

As I said earlier, DDD is nerd heaven, and while this approach serves the nerd worldview, it leaves the end user in the cold. If there is any great tragedy in the history of DDD it is that its notion of *language* has been misappropriated for overly low-level constructs instead of celebrating the business vocabulary, as the decades-old legacy of domain analysis has done. It reminds me of a keynote that Guy Steele gave at OOPSLA about the granularity of expression of language. His talk was also about language; in his case, it happened to be Java. I think the same critique might apply at one higher level of design discourse, and we find that problem in DDD. The Whole is missing and, as Rebecca Wirfs-Brock has said, an object-oriented design is more than just a bag of objects.

By, of, and for the people

I went to a workshop session at the DDD conference where people were talking about software that helped in the maintenance of factory floor machinery. I decided to apply an old test. I sat and quietly listened to their conversation and observed how long it would take for me to find out what business they were in. They were discussing their ubiquitous language, so I thought: This should be easy, I listened. The talked about *entities* (which I figured out were things like machines) with *components* (motors, belts, bearing assemblies) and *associations* (pipes, tubes, and wires connecting the machines) and so many other nerd-centric concepts that I eventually had to ask what these machines actually built. But *motors*, *wires*, and specific *machines* were mentioned only in passing as though such use would somehow sully or corrupt design understanding. It was design without analysis. It is not a long jump from there to say that it was design without adequate thinking.

Several months later I would travel to China to serve a client that wanted advice on DDD. They told me they had contacted Eric but that Eric really didn't want to travel to China; besides, he had written a book that conveyed enough of his knowledge that he didn't have to be there in person. I sat

through hours of PowerPoint® presentations of UML diagrams of design: classes, class hierarchies, and associations. Most of the workers were 20-somethings with a few months or, occasionally, a few years of experience. There had been no concerted effort to canvass the market, to make a domain model (i.e., an enumeration of the business domains and an understanding of their relationships). They could not tell me what problem they were solving. It was nerd heaven.

To make analysis work means mixing with the hoi-polloi: of getting one's fingers dirty and understanding the non-hierarchical messiness of the real world. DDD structure is a hierarchy (or, at best, a directed acyclic graph) with perfect symmetry. Its symmetry gives it elegance, and that elegance appeals to the analytical mind. Our mind uses the approximate symmetry of nature to cheat how memory works, to save energy: when I meet you, my mind stores away an image of only half your face. Yet if you carefully analyze a picture of yourself, you'll find you're not symmetric. (Take a picture of your face, straight on; make two copies, and cut each copy into perfect halves, right down the nose. Put the halves together in various ways to see the many versions of You.) Yet architect Christopher Alexander reminds us that beauty owes to the breaking of symmetry and to harmonizing with the inevitable imperfections of nature. Many, such as Jef Raskin (*Humane Interfaces*), admonish us to create software tolerant of human frailty and errors.

Mental Models

One negative, to me, is that DDD talks very little about people. Most software today interacts with an end user, and it's crucial that the underlying software structure, the elements of the human interface, and the end-user mental model be well-aligned. DDD seems like the nerds' revenge, sitting inside the machine, unbothered by human beings.

If the mental model could steer or more directly drive how we create bounded contexts, I think we would end with systems that have fewer bugs, that evolve better, and that are easier to understand. These concerns are often far from the design considerations that one can express in a programming language or even in a DSL in its new sense. They are an issue of analysis and model-building that long precedes even the selection of a programming language.

The new house I am building will lie in nature amongst sand dunes close to the sea. We talked to many who have built homes in the area about how their layout deals with the usual blustery weather and occasional sandstorm; how the foundations will work with the sand-and-turf layer soil; how to take advantage of the incredibly unique light for which the area is world-famous, and so forth. It wasn't just looking at house floor plans, picking one that was spacious and easy to clean and plopping it down on our property. I bought a drone and we did aerial surveys. We prototyped several layouts on our lot. None of these things would have come from the architect's drawings or from his drafting tools. And so it is in software. It is critical to tie the overall system form, as well as the design of its parts, to the mental model of the user. DDD talks about objects: getting the end user mental model into the software is the whole reason OO was created. DDD talks about patterns: letting the inner feeling of the *habitants* of the dwelling attune their instincts and to drive the design is the whole Alexandrian agenda.

Break the Symmetry

At the DDD conference I talked about a renaissance in object-oriented programming called DCI: the Data, Context and Interaction paradigm. It tried to return to Alan Kay's fundamental ideas beneath what he named object-orientation years ago. Alan's world was the world of children's mental models and representations of the world. It is a messy world. Things don't necessarily fit well into neat encapsulations of highly coupled data and methods. While such loci (called objects) are a useful approximation, the whole story unfolds in the networks of interaction between them. And that's where the value lies.

In Kay's world, some of these interactions happen on-the-fly as objects find opportunities to collaborate; others can be designed, say, as use cases. In DCI we have admittedly focused on the latter. In both cases the focus is on objects and their interactions. Objects exist to interact. Classes exist to constrain your thinking to a bounded realm of computation. It has no value on its own, but is simply a context within which some computation unfolds in a way that engineers can reason about. One might call them bounded contexts and, from a cognitive perspective, classes and the suitably named bounded contexts of DDD share this property. They are a locus of a consistent worldview: a consistent vocabulary with self-contained consistency.

DDD has hierarchy at its heart. This has implications: a given domain — an area of application or design discourse — may not find expression in the code but may in fact span multiple bounded contexts. This means that to reason about changing a domain requires coordinated changes to multiple modules, as there seems to be no way to separate bounded contexts and modules in DDD. This, of course, goes against half-century old wisdom about the importance of coupling and cohesion.

A (bounded) context in DCI is a use case: a network objects that can be configured to rise to the occasion to carry out some computation that reflects an *operational model* from the end user's mental model. An object may participate in several such contexts. Classes lack this notion of connection or of belonging to a greater whole: we focus only on how they are allowed to talk to their neighbors. And much the same is true of bounded contexts.

DCI can also classify software by such loci of domain considerations; it just uses the class, which has longstanding stature in doing exactly this. The dynamic relationships between objects are in some sense in another dimension. And in yet another dimension we find that there are whole sets of subdivisions within a domain that can work interchangeably in an operational model. Real software design is complex, and a design is elegant and maintainable to the degree that it can express as many dimensions as there are in the application. If you look at the major advances in software architecture and design, they have been about how well they can express multiple degrees of symmetry elegantly. Aspects tried to express multiple degrees of symmetry but were inelegant. C++, Ruby, and C# are all in some way messy languages that can express many geomerties of software form; Scheme, Lisp, and Smalltalk are elegantly symmetric. Guess which ones survived. DCI is consciously rich in broken symmetry, while DDD is almost purely hierarchical. There is essentially one building block: the bounded context.

Eric's book often uses patterns as expository structures. This is one case where I believe the theory could be priceless in moving to the next stage, Pattern theory is all about interaction between and

not-separateness of the things we create — something that bashes symmetry and hierarchy against the rocks. Alexander himself uses the phrase "not-separateness" often, and is known for his essay "A City is Not a Tree" that decries urban planning based on delineated neighborhoods — their bounded contexts. Patterns go beyond DCI in that DCI only breaks two (or maybe three) dimensions of symmetry. They are multitude. The designer who ignores them will produce an inhumane system.

Many idiomatic language constructs are a form of symmetry breaking. Copying an array in C, a novice would do like this:

```
char array1[10], array2[10];
. . .
for (int i = 0; i < 10; i++) {
   array1[i] = array2[1];
}
```

But that is just syntax and semantics compliant with (symmetric to) the canonical grammar of the language., If we mix several lingistic constructs, we get:

```
char array1[10], array2[10];
. . .
char *cp1 = array1, *cp2 = array2;
while (*cp1++ = *cp2++);
```

True richness of expression comes in concept overlap. For example, in flight reservations, there is a broad overlap between routes and airplane types and therefore classes of service. DDD separates them because, well, it has only one dimension of symmetry along which to express such concerns, so the delineation is linear. A hierarchy is barely more than a linearization. Though no linear sequence has a unique hierarchy, I can map every hierarchy to a unique sequence. I cannot do that with a directed graph, or even a directed acyclic graph, or a graph. Complex designs are graphs. There are many arguably right ways to slice the pie, yet the designer must pick just one — otherwise, bounded contexts would not be bounded, for example.

I think there is a potential huge win in a next-generation DCI that can build on some DDD notions, or a next-generation DDD that can express multiple hierarchies. It's low-hanging fruit and has already proven as a way of slicing hierarchy in a way that aids human design effectiveness, as the research by Héctor Valdecantos et al. has shown. The technology exists, and combining the two is largely an informed engineering exercise.

Domain Specific Languages

We must make the ubiquitous language ubiquitous. There is power in language, and there is a danger of the good driving out the perfect if we view DDD as the final word on the linguistics of design structure. In fact, DDD overly dilutes its aspirations for design linguistics relative to prior art, and

both are likely a shadow of what might be possible. Alexander resurrected the longstanding notion of design languages in his pattern formalism; perhaps, together with the psychological foundations of object-orientation, we can open a new door of insight on how to bring the broad notion of language to the fore of technology. We have *S* for statisticians, *Excel* and *yacc* and *PowerPoint* for their respective domains, yet there may be hope of raising the bar to create a language for automobile navigation software or video game design. To do so will probably displace software design from its comfort zone among its foundations in arithmetic. Most languages still have the fundamental operators of algebra and assignment, but these have little to do with most of the problems of automated assistance of human tasks. In such a world, all a design method can do is to modulate the vocabulary of identifiers, *Ubiquitous language* — at the level of *language* rather than vocabulary — is a paradigm shift into which we as an industry have not yet waded. It is an exciting possibility.

A Taxonomy of Theory and Heuristics

Too often computer scientists seek overly deep rationalizations or overly formal reasons for their code. We can analyze user needs to death. But as Freud says, sometimes a cigar is just a cigar.

Sometimes disjoint bounded contexts are just the ticket. While nothing in the universe is perfectly separate from anything else, sometimes the separation is good enough that each concept can have its own self-contained entity. There is a time and place for such separations, perhaps at the grossest level of scale (separating the missle navigation software from the program that controls the hydraulics of the radar antenna) and the finest (a String is a pretty high-integrity concept.)

Yet things are messy in between, and the symmetry breaks. We need both heuristics and design formalisms to decide when to create new entities and when to use more sophisticated approaches such as DCI. Sometimes culture wins out and what is in reality an entangled concept deserves its own identity because of how it matches human mental models. Perhaps such situations should cause us to cast Occam's Razor aside for the sake of pragmatics and the power of software maintainability that comes with comprehension. A true story is useless if it cannot be understood, and even untrue stories (fables, myths) convey great power.

Sendoff

So there you have it: a rambling mind dump of ideas, hopes, and heresies related to our shared aspirations for design. I hope it brings a smile to your face, and gives you ponder about something on which to work for the next 15 years. Godspeed.

Neighbors, J. (1984), "The Draco Approach to Constructing Software from Reusable Components," IEEE Transactions on Software Engineering 10, 5, 564–574.

Siniaalto and Abrahamsson. "Does Test-Driven Development Improve the Program Code? Alarming results from a Comparative Case Study." *Proceedings of Cee-Set 2007*, 10 - 12 October, 2007, Poznan, Poland.

Guy Steele. Crafting a Language, OOPLSA 1998.[58]

James Coplien. *Multi-Paradigm Design for C++*. Reading, MA: Addison-Wesley, 1998.

Christopher Alexander. "A City is Not a Tree." In *Architecture Forum*, Sustasis Foundation, Boston, MA, 1965, ISBN 978-0-98-934697-9.

Angela Martin, R. Biddle and J. Noble. "The XP Customer Role in Practice: Three Case Studies." *Proceedings of the Second Agile Development Conference*, 2004.

Héctor Valdecantos, Katy Tarrit, Mehdi Mirakhorli , and James O. Coplien. "An Empirical Study on Code Comprehension: Data Context Interaction Compared to Classical Object Oriented." *Proceedings of ICPC 2017*, IEEE Press, May 2017.

Taichi Ōno. *Toyota production system: Beyond large-scale production*. Cambridge, MA: Productivity Press, 1988.

Jef Raskin. *The Humane Interface: New Directions for Designing Interactive Systems*. Reading, MA: Addison-Wesley, 2000.

[58] https://www.youtube.com/watch?v=_ahvzDzKdB0

Free the Collaboration Barrier - Mel Conway

1. DDD Prototyping: Extend the tech-business collaboration

2. Introducing the code-free API

3. Proposal: Release the Visual API into the wild

Plan to throw one away; you will, anyway.

Frederick P. Brooks, Jr. “The Mythical Man-Month” {/blockquote}

1. DDD Prototyping: Extend the tech-business collaboration

What is the Collaboration Barrier?

The collaboration barrier is the moment on the timeline of a business application development project at which the major contribution to the end result shifts decisively toward the software developers and away from the business members of the design team.

The collaboration barrier typically occurs when enough is understood about the domain model that coding in earnest can begin. At that point there is confidence that the likelihood of disruptive change of ideas has become acceptably low, so the ideas that have been captured in the domain model can begin to be frozen in code.

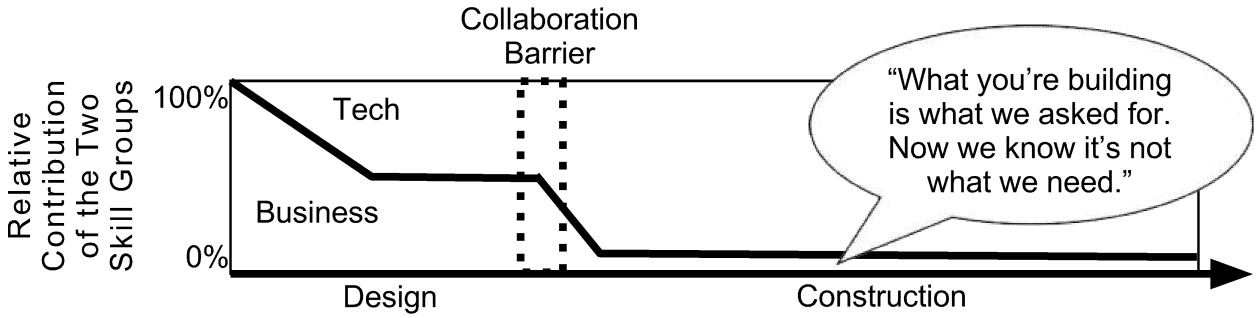


Figure 1 - The contribution of business people drops when coding begins

Figure 1 shows a purely qualitative model of how the technical people and the business people share participation in determining the final outcome as the project progresses. There is a kink to the left of the collaboration barrier. Before this kink the business people are educating the tech people about the business. After the kink they have developed their ubiquitous language and are describing the domain model fully collaboratively. To the right of the collaboration barrier the contribution of the business people to the final product becomes minimal.

The Cost of Change

This can change if the team discovers that they must go back and revise the domain model. After the collaboration barrier the cost of actually going back and changing the domain model begins to rise, and it continues to rise as time progresses and more work must be undone. In other words, before the collaboration barrier the cost (in terms of ultimate project delay) of changing your mind is low and flat. After the collaboration barrier the cost typically rises monotonically. So after the collaboration barrier, if the realization occurs that there is something about the design that needs to change, the decision process may turn into a value tradeoff between product quality and project schedule.

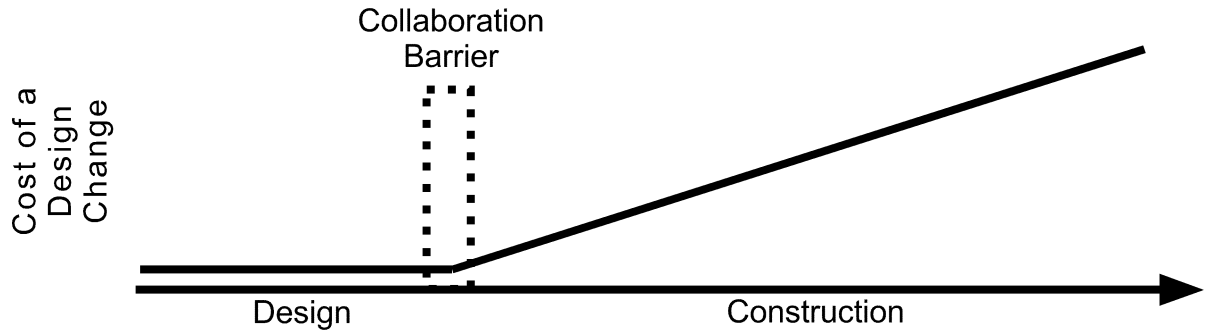


Figure 2 - The more code that has been committed, the greater the cost of change

Figure 2 shows, again qualitatively, the cost of change as time progresses. This cost is some combination of schedule delay and quality loss, depending on how the change is handled.

Obviously you want to get the design right before the collaboration barrier. The conundrum, in particular if this is a first implementation of the requirement, is that (to put it starkly) *you don't know what to build until you've built one*. IBM's experience building OS/360 led to Fred Brooks's lesson at the top.

The Case for Business Participation in Prototyping

The answer, then, is to build something cheaply and quickly that you can use in order to learn what you really need to build. I'm calling this thing a *prototype. The goal of a prototype should be to maximize learning at a controlled cost.*

My premise here is that, in order that this opportunity to learn be maximized, the business people must be fully engaged in building the prototype. If this is the case, the effect of introducing a prototyping stage into development is to shift the collaboration barrier to the right, as shown in Figure 3.

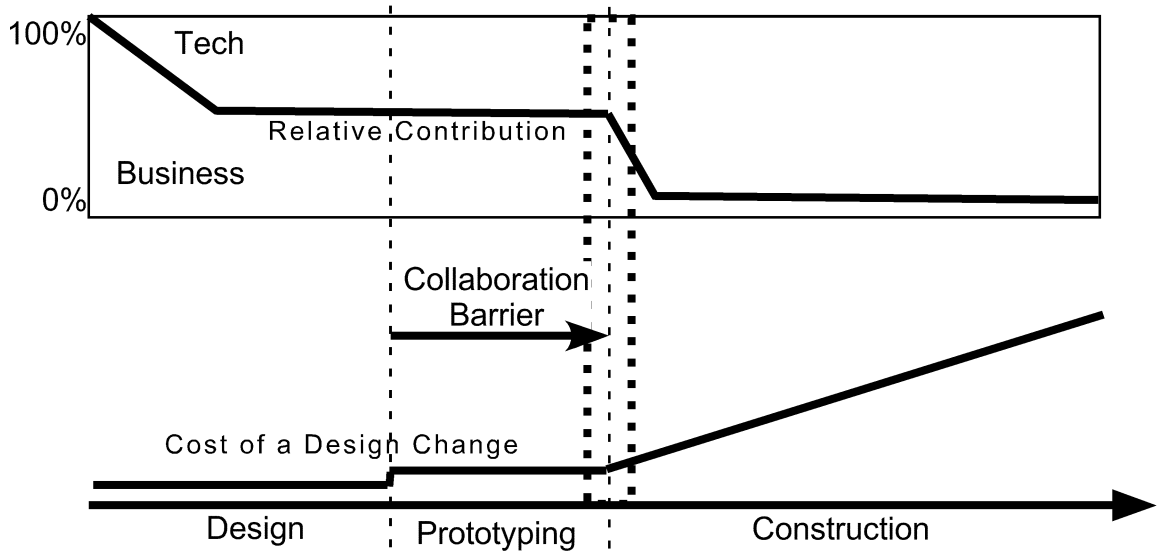


Figure 3 - Engaging business people in prototyping moves the collaboration barrier to the right

If we accept that the prototype is a throwaway (and there is no reason at this point to doubt that) then the effort of building the prototype does not contribute to the final code. This effort is "lost", and it adds to the total cost of the project. What does this cost buy? To quote Brooks:

> "The management question, therefore, is not ***whether*** to build a pilot system and throw it away. You ***will*** do that. The only question is whether to plan in advance to build a throwaway, or to promise to deliver the throwaway to customers".[59]

I am taking the liberty of interpreting Brooks as follows:

> What building the prototype buys is avoiding having to scrap and rebuild the delivered system, or more optimistically, it buys building a more usable system the first time. What we have learned since Brooks is how to throw it away in little pieces instead of all at once.

Why can prototyping in which the business people are fully engaged lead to a more usable system? Because the business people, who are the only available prelease resources for measuring usability, are right there participating in its construction. They are thinking about usability all the time.

So we have begged the question: how can we build the prototype in a way that the business people, who we assume are not programmers, are fully engaged in the process?

Partition the Work

The answer I propose here is to divide the prototype into a part built by the developers and a part built by the business people. Here are some requirements on this approach to dividing up the task.

[59]Brooks, Jr., Frederick P. (1975, 20th Anniversary Edition, 1995). "The Mythical Man-Month". Addison-Wesley

- The business people must have control over use of the prototype so they can show it to their colleagues. The part they build must be lightweight in order to permit experimentation, so they can learn from experience and modify the design, and it must not require coding skills.
- The parts that the business and technical people build must correspond to the respective expertise and experience that these two groups bring to the table. Specifically, the business people should work at the level of use cases and the technical people should work at the level of domain objects.
- Change is inevitable as learning occurs. This learning should be reflected smoothly in the evolution of the prototype. (Ideally, as learning occurs, the prototype should evolve toward a small monolithic version of the ultimate system.)
- The sizes of the respective parts, the interface between them, and their relative rates of change must be such that the developers and business people can continue to work concurrently.

The Two-faced Prototype Model

The two-faced model (Figure 4) is an environment combining traditionally-built domain objects and pictorial use-case descriptions built by non-programmers. The Visual APIs enable these two parts to work together synergistically while maintaining the benefits of each.

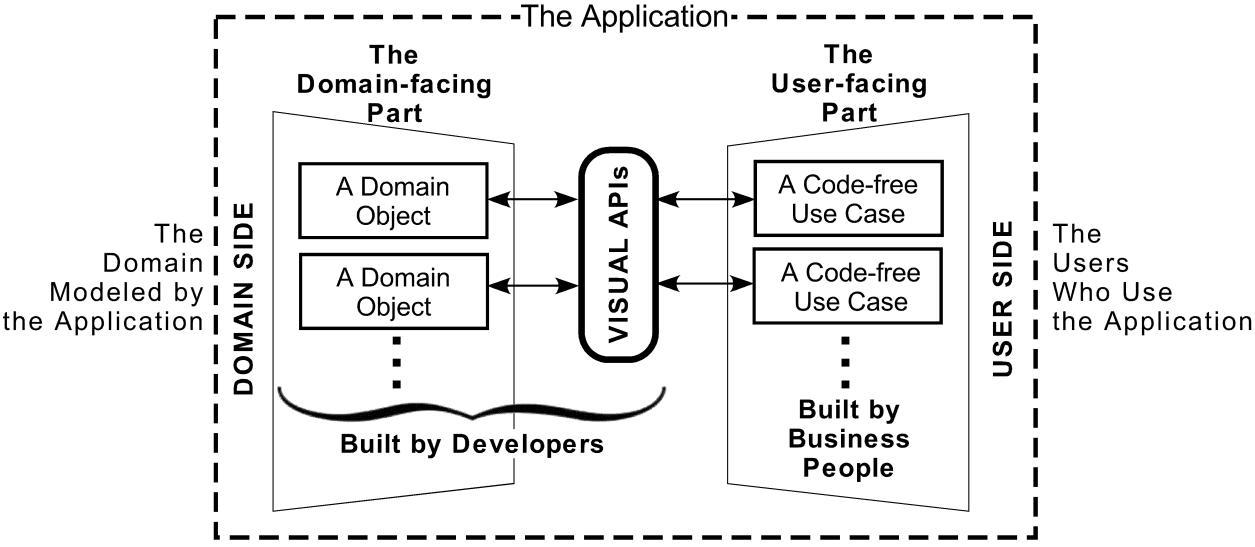


Figure 4 - The two-faced prototype model

Code-free Use Cases

The code-free use cases execute value-delivering user interactions. They do more than simply describe a user interface because they communicate with domain objects, work with values derived directly from them, and respond appropriately to user events, possibly with state changes.

I have developed a unidirectional object-flow wiring language that seems to be sufficent for describing use cases according to the bulleted list of requirements above. I do not assert that it is the only solution, but it is an existence proof.

This wiring language is not powerful by itself, but it is seamlessly extended to encompass the business processes of the domain objects through the Visual APIs. Wiring is a pictorial, non-procedural flow language that doesn't do arithmetic or even the simplest algorithms involving looping or branching. It occupies a sweet spot between power and accessibility (that is, availability to non-programmers). It does not need to be made more complicated because, wherever more power is required such power is domain-specific, belongs in domain objects, and is accessible through Visual APIs, described in the following section.

Figure 5 illustrates how the wiring language fits into the power-accessibility tradeoff. It is capable of handling functional composition (by connecting wired components) and collections (business objects that flow on the wires), rendering user interfaces, responding to user events, and little else. It has been my observation that much of what happens in these wired use-case descriptions is assembling and disassembling collections. The objects that flow down the wires can be complex, for example, records whose elements have domain-specific behaviors.

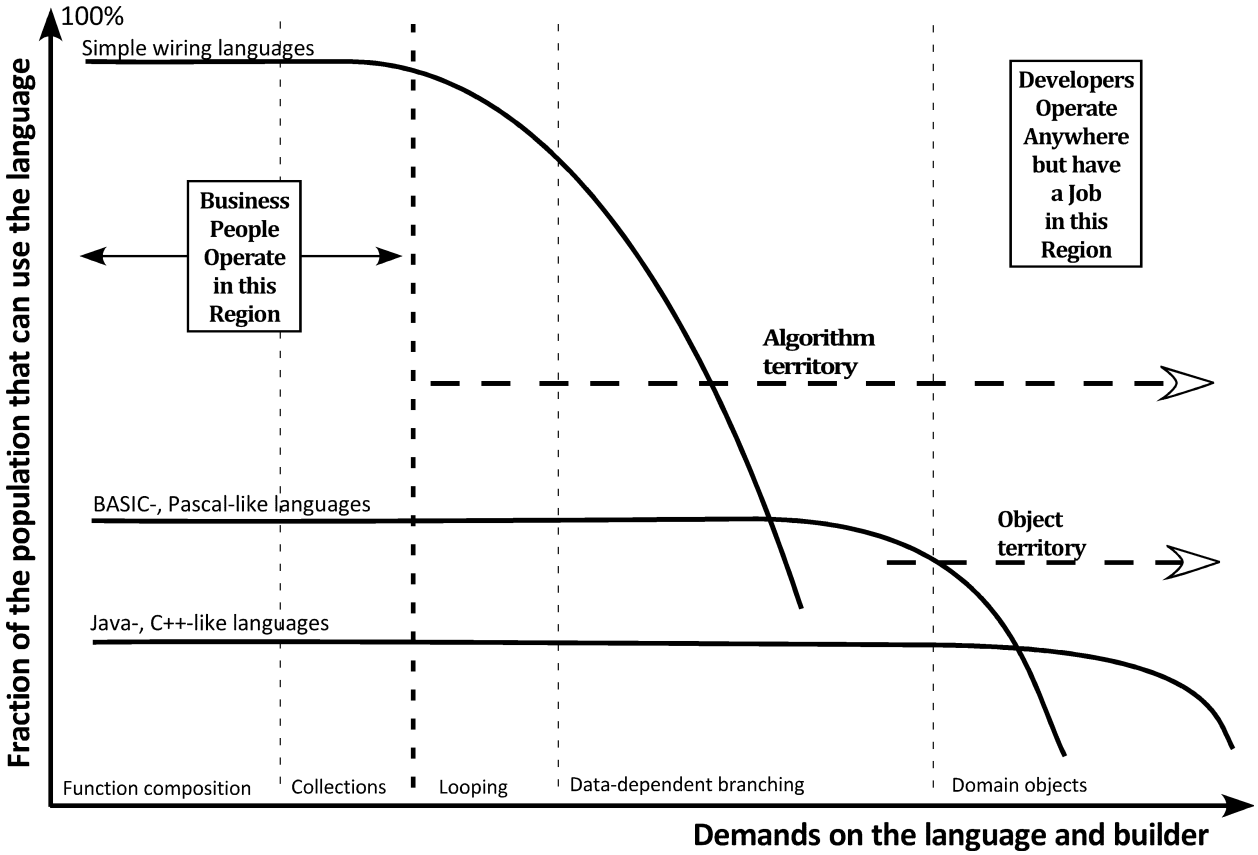


Figure 5 - Simple wiring languages are less powerful (by themselves) but more accessible

Visual APIs

The Visual API replaces the demand on the builder to construct grammatical text by presentation of choices presented in dialog windows. It enables the business people to access domain objects and parameterize messages to them without leaving their comfort zones. These messages are sent by

wired components, and their return values flow down the wires.

Figure 4 shows that Visual APIs are built by developers. The payoffs from this work are the ease of communication between the technical people and business people, and *extending the time that they are fully engaged further into the development process.* These are the business benefits from prototyping of the two-faced model.

2. Introducing the code-free API

This section combines multiple existing distinctions into one conceptual framework, in order to extend this framework. Figure 9 puts the concepts together in one place.

A. The Symmetry/Flexibility Distinction

Each API has two aspects.

- It is a ***formal specification*** of a request-response interface through which a ***client*** software component makes a request of a ***server*** software component.
- It is a ***contract*** to conform to the formal specification between two communities of developers. Developers in the ***producer role*** build conforming servers; developers in the ***consumer role*** build conforming clients. Somebody publishes the API; often, but not always, it is a producer.

Once the API is published and starts being used it might or might not be acceptable to change it, depending on the relationships between producers and consumers. This distinction divides APIs into two classes, *negotiable* and *fixed*; see Figure 6.

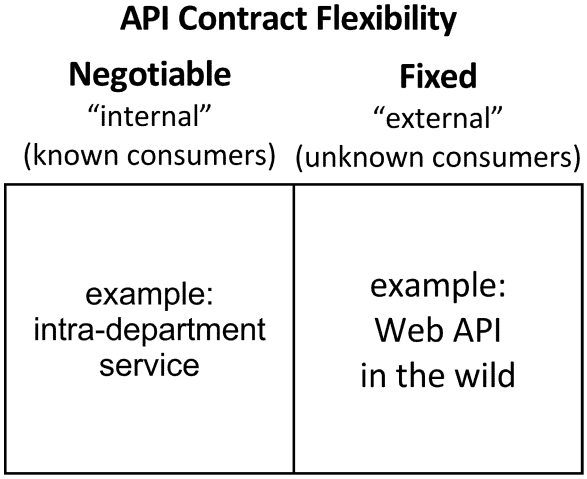


Figure 6 - The API contract might be negotiable or fixed, once the API is in use

This corresponds to the distinctions in the general API literature. This literature almost always assumes that both producers and consumers are programmers, or at least can work with the technical details of the formal specification. However, the discussion of DDD prototyping above contradicts this assumption because the consumers, the business people, are not usually programmers. So we need to turn this 1x2 into a 2x2; see Figure 7. The "symmetry" dimension of this array corresponds

to the specification aspect of the API; the "flexibility" dimension of the array corresponds to the contract aspect of the API.

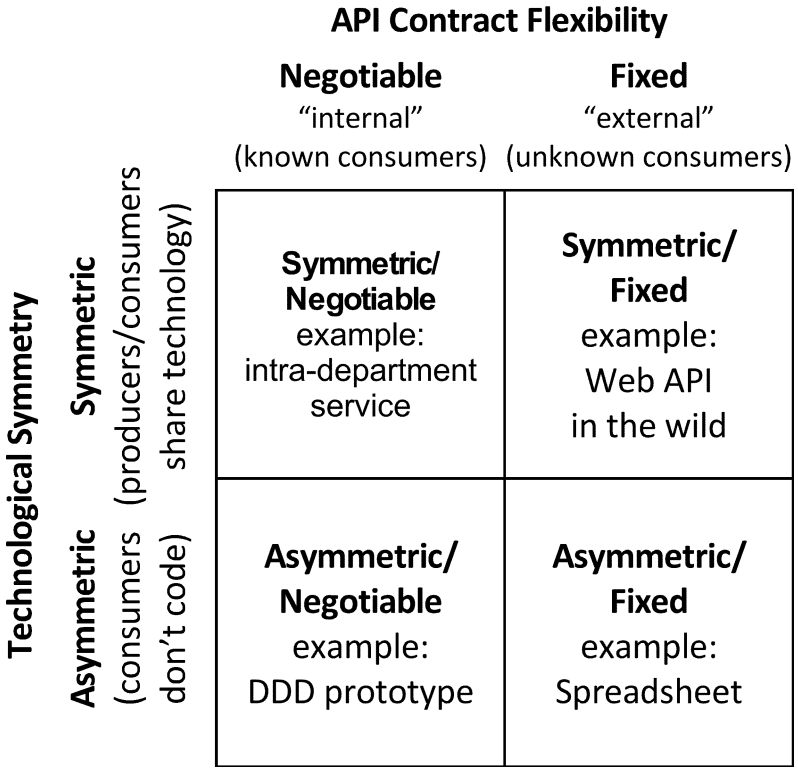


Figure 7 - The Symmetry/Flexibility API diagram

Note that the spreadsheet, which has been recognized for a long time as some not-readily-classified species of programming language, falls into this classification scheme. A spreadsheet can be seen as an API, as follows:

- The consumer is the spreadsheet's user.
- The producer is the vendor, for example, Microsoft/Excel or Lotus/1-2-3.
- The formal specification is the *application model*, stated here approximately:
  - There is an expandable two-dimensional array of *cells*, addressable by row and column. Each cell contains an *entry*.
  - Each entry can be a literal or the value of a function whose definition is also part of the cell, but is usually invisible.
  - The arguments of the function are literals or cell references.
  - A request occurs every time the user changes an entry. The response is a network of constraint-resolution calculations that attempt to restore consistency among the entries, which have presumably been rendered inconsistent by the change. Sometimes this attempt will fail and an error message will appear.

Why mention the spreadsheet here? Because describing its application model will allow us later to segue to the two-faced model.

B. The Construction/Operation Distinction

There are two distinct stages in the development of any unit of software, the ***construction stage*** and the ***operation stage***. In order for us to understand fully what a Visual API is, these two stages need to be brought into the same conceptual framework and regarded as two aspects of the same thing: the ***life-cycle*** of that unit of software. For simplicity we limit this discussion to the development of a client-server interaction according to an API.

- During the construction stage the producers and the consumers each work with their respective tools building response code and request code, respectively, according to the contractual aspect of the API. Their tools build the respective codes to conform to the formal specification aspect of the API.
- During the operation stage the producers and consumers, and (usually) their tools, are not in the picture. Typically, the request code operates on some device and initiates an interprocess communication, which initiates the response code on some, possibly different, device.

The life-cycle/role diagram of Figure 8 captures this distinction for the Visual API. Notice that there is a new element: the Public VAPI Posting. This is visible to the consumer's tool. *It manages the interaction with the consumer; this interaction generates and stores the request code in the client.* All this interaction and code-generation capability is built in advance by the producer; this effort is an extra cost whose benefit is the simplicity experienced by the consumer.

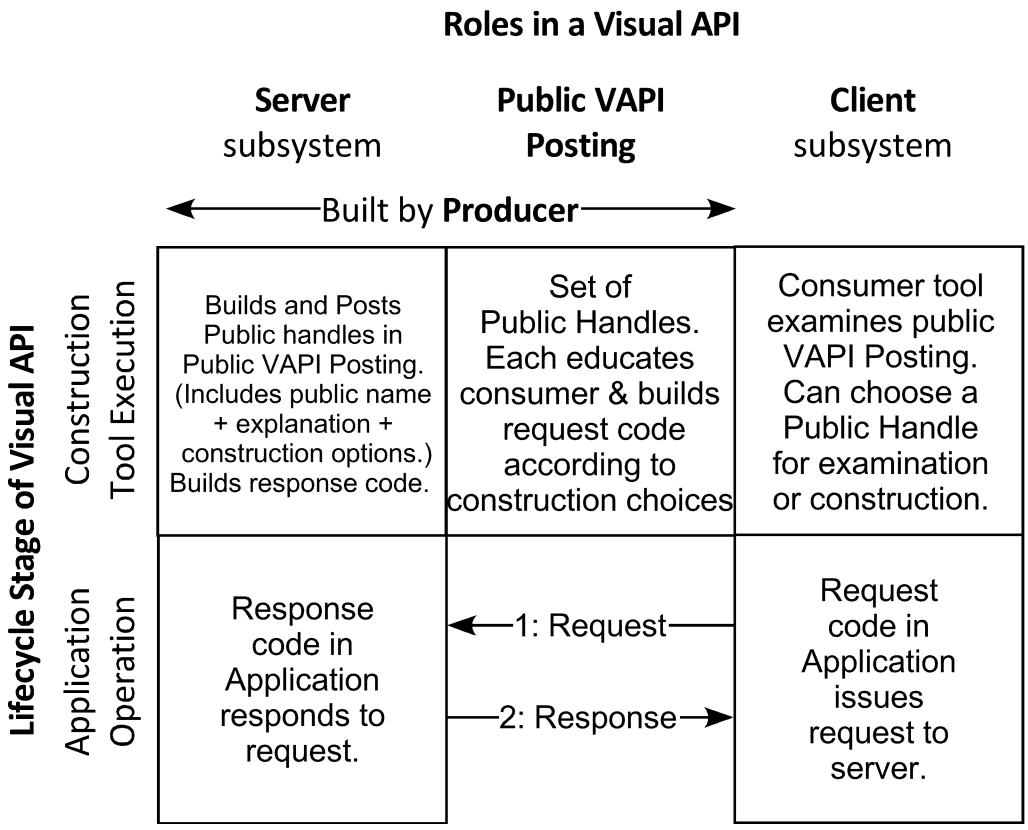


Figure 8 - The life-cycle/role diagram of a Visual API

Figure 9 summarizes the concepts presented so far.

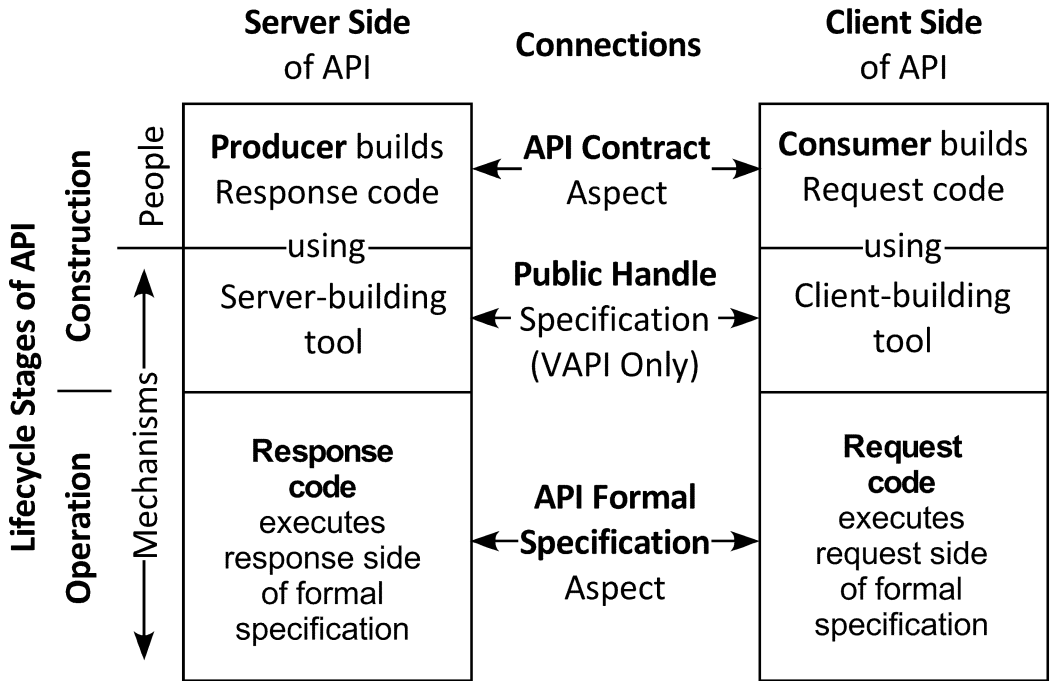


Figure 9 - A Summary of the Concepts

3. Proposal: Release the Visual API into the wild

Product Concept

Now let us revisit Figure 7 replacing the spreadsheet example by an example that doesn't exist yet. I'll call it a do-it-yourself app builder, "DIY App Builder" for short. It belongs in the Asymmetric/Fixed quadrant of Figure 7. That is, its API contracts are not negotiable, and its clients can be created by non-programmers. Figure 10 simply replaces "Spreadsheet" in Figure 7 by "DIY App Builder".

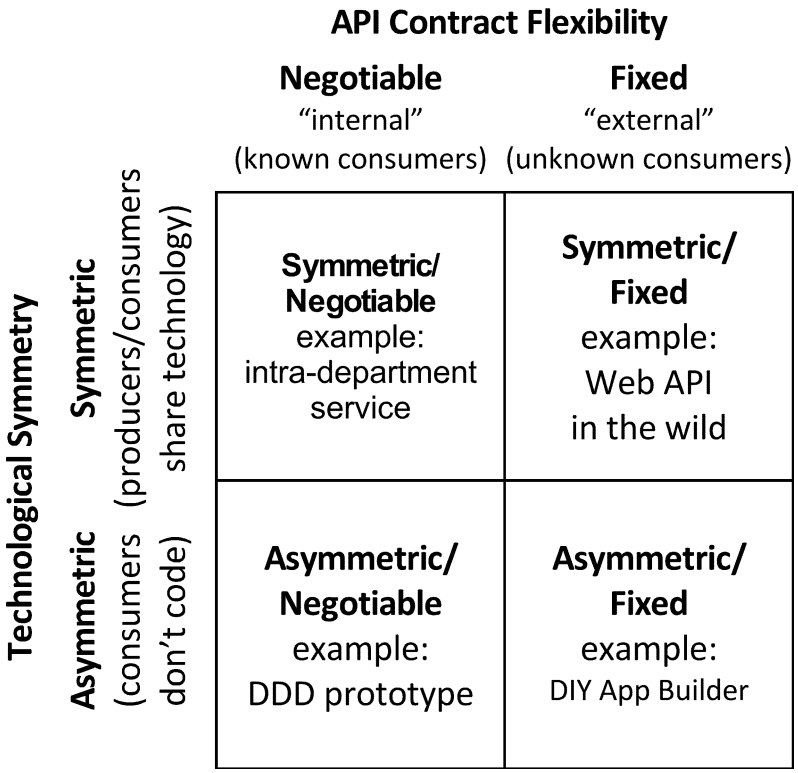


Figure 10 - The DIY App Builder Replaces the Spreadsheet in Asymmetric/Fixed

What is a DIY app builder? Think of it as something that builds business applications that conform to the two-faced model (Figure 4) and that don't require that the consumers of its APIs be programmers. The vision for DIY app builder APIs is that they can be released into the wild the way Web APIs and spreadsheets are; they will allow nonprogrammers to build certain classes of business applications. What this looks like is the subject of the next section.

Figure 11 portrays my conception of a DIY App Builder product.

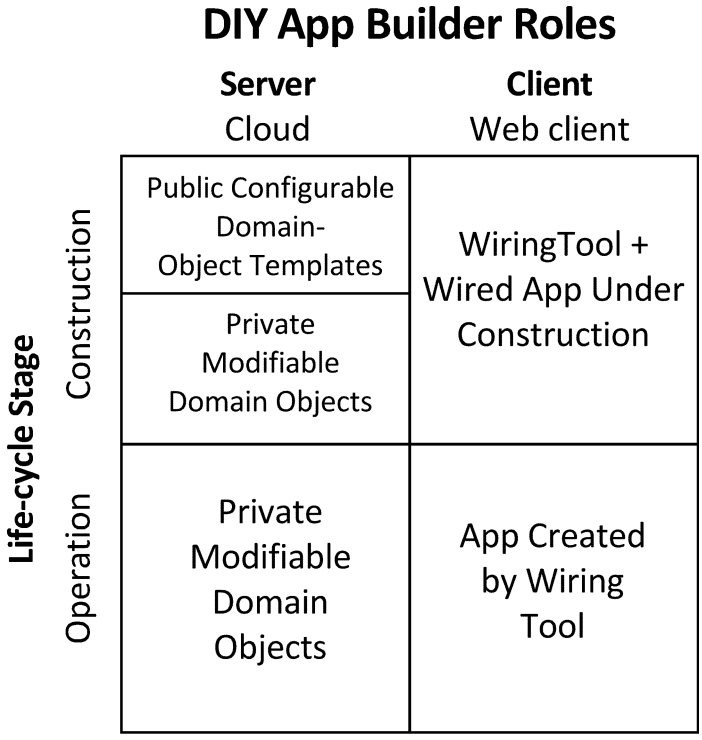


Figure 11 - The life-cycle/role diagram of the DIY App Builder

- **Client role**. Applications are built in a web browser and, initially at least, they are also executed in a browser. The user's construction tool is a wiring tool that runs in a browser concurrently with the application it is building.
- **Server role**.
  - **Construction stage**. In Figure 11 "Public" means available to everybody; "Private" means available to a particular consumer account.\ The consumer builds applications based on domain objects that have been configured from a publicly available library of domain-object templates. These templates cover a range of small- and medium-size business objects that have already been abstracted by the creators of specified-function small-business software packages now available. (The abstraction process has been done; now it needs to be adapted to the two-faced model.) The consumer can operate in two modes.
    * Adapting a public template for specific use as a private modifiable domain object.
    * Wiring an application that accesses the consumer's repertoire of private modifiable domain objects through its visual APIs.\ What does "modifiable" mean? This is not yet clear, and is probably context-dependent. The intent is to maximize the consumer's opportunity to change his/her design even after operation has begun.
  - **Operation stage**. These are the same private modifiable domain objects.

Implications

This concept might well be limited, at least initially, to the small/medium business software market. One way products are delivered in this market is through local consultants who customize existing

packages for their clients. A few examples include: retail cash register/inventory/ordering, non-profit donor management, and client project management/billing. This product should enable these value-add consultants to be more productive and to build proprietary domain objects that can increase their added value. Some small fraction of business users might venture into building their own applications.

If this product is managed appropriately it will build a community of users who will interact and support each other. In the long run I can imagine the community of Free and Open-Source Software, now mostly limited to developers, splitting into two branches (corresponding to the two parts of the two-faced model): the existing developer branch and a new end-user branch, whose members trade encapsulated wiring diagrams. Some number of the developers in the developer branch will build domain object templates for this product, enlarging its market.

7 Years of DDD: Tackling Complexity in a Large-Scale Marketing System — Vladik Khononov

One morning, back in 2010, I got a phone call from a friend. He said he was starting a new company. The business was not going to be simple, but if I joined him, from the technical perspective I could do whatever I wanted. I agreed to join him, just like that!

In this essay I'd like to share what I got myself into. In particular, the Domain-Driven Design (DDD) side of our company's story, since we employed this methodology from day one.

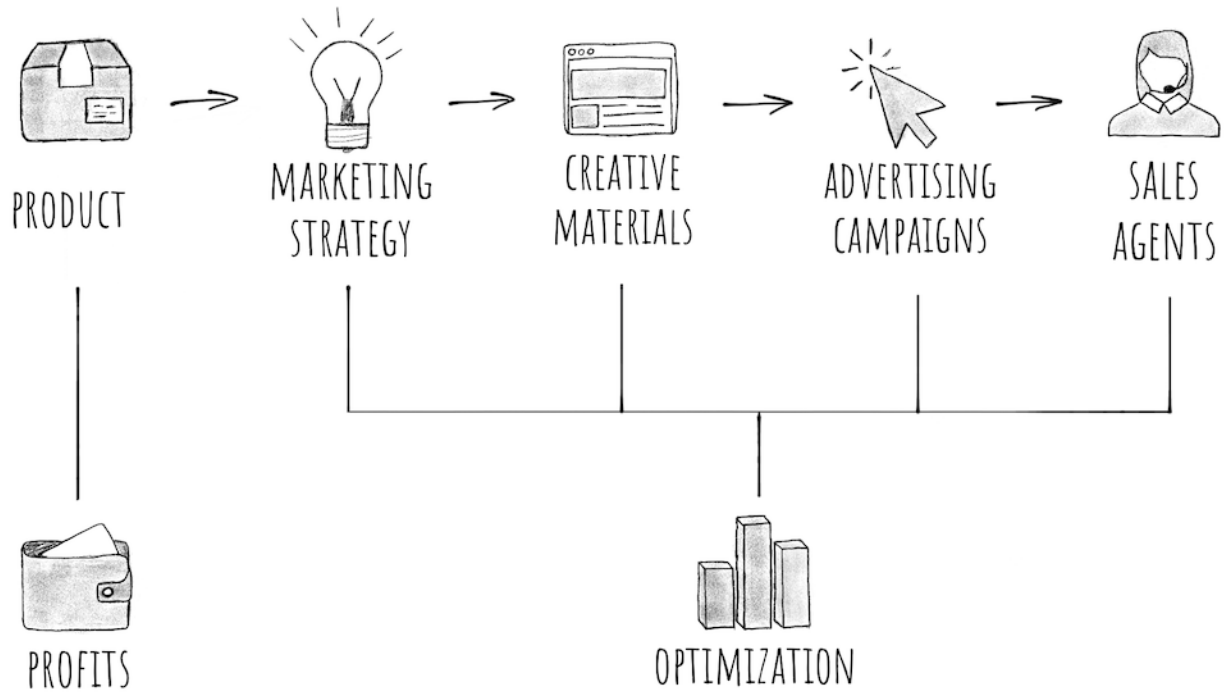
I'll start with a walk through the stories of 5 Bounded Contexts that demonstrate the different approaches to DDD we tried, and the results of our efforts. In the second part, I will use those 5 stories to share some practical advice on Domain-Driven Design, domain modeling, event sourcing, CQRS, and microservices, all based on our experience at Internovus.

But first, as well-behaved DDD practitioners, we'll start with the business domain.

Internovus

Imagine you are producing a product or a service. Internovus allows you to outsource all of your marketing-related chores. We will come up with the best marketing strategy for your product. Our copywriters and graphic designers will produce tons of creative materials, such as banners and landing pages, that will be used to run advertising campaigns that promote your product. All the leads generated by these campaigns are handled by our own sales agents, who will make the calls and sell your product.

Most importantly, this marketing process provides many opportunities for optimization, and that's exactly what our analysis department is doing. They analyze all the data to make sure our clients are getting the biggest bang for their buck, be it by pinpointing the most successful campaigns, the most effective creatives, or by ensuring that the sales agents are working on the most promising leads.



A New Hope

Since we were a self-funded company, we had to get rolling as fast as possible. Therefore, for the first version, we had to implement the first third of our value chain:

- A system for managing contracts and integrations with external publishers;
- A catalog for our designers to manage creative materials; and
- A campaign management solution to run advertising campaigns.

The latter meant not only an information management system, but an advertisement serving and hosting solution as well. To be honest, I was overwhelmed and had to find a way to wrap my head around all the complexities of the business domain. Fortunately, not long before we started working, I got a book that promised just that. It described a way to tackle the complexities at the heart of software: Domain-Driven Design.

Surely The Blue Book reads like poetry[60], but it is not an easy book to read. Luckily for me, I got a really strong grasp of Domain-Driven Design just by reading the first four chapters.

Guess how the system was initially designed? — It would definitely make a certain Kazakhstan based, prominent individual from the DDD community very proud…

[60] https://www.infoq.com/interviews/jimmy-nilsson-linq

@DDD_Borat

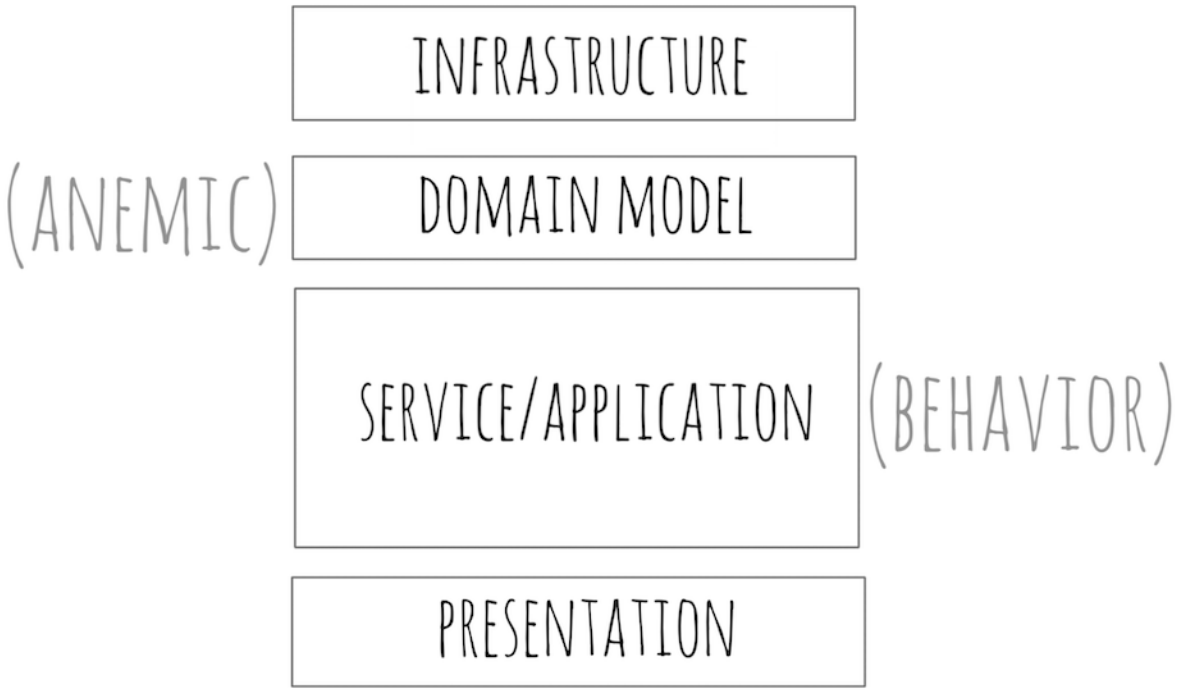
Part I: 5 Bounded Contexts

The First “Bounded” Context

The architectural style of our first solution could be neatly summarized as “Aggregates Everywhere”. Agency, Campaign, Placement, Funnel, Publisher - each and every noun in the requirements was proclaimed as an Aggregate.

All those so-called aggregates resided in a huge, lone, bounded context. Yes, a big scary monolith, the kind that everyone warns you about nowadays.

And of course, those were no aggregates. They didn’t provide any transactional boundaries, and they had almost no behavior in them. All the business logic was implemented in an enormous service layer. A typically anemic domain model.



In hindsight, this design was so terrible it resembled a by-the-book example of what Domain-Driven Design is not. However, things looked quite different from the business standpoint.

From the business's point of view, this project was considered a huge success! Despite the flawed architecture, and despite our unique approach to quality assurance — QA is for cowards — we were able to deliver working software in a very aggressive time to market. How did we do it?

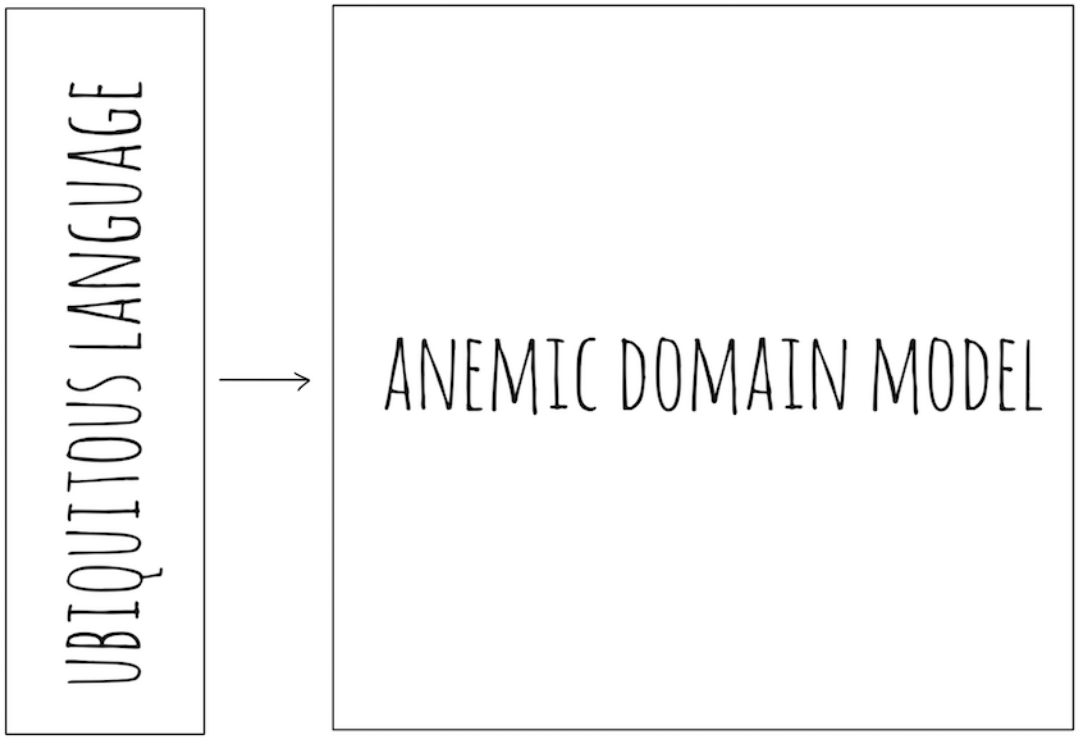
A Kind of Magic

We somehow managed to come up with a robust Ubiquitous Language. None of us had any prior experience in online marketing, but we could still hold a conversation with domain experts. We understood them, they understood us, and to our astonishment, domain experts turned out to be very nice people! They genuinely appreciated the fact that we were willing to learn from them and their experience.

The smooth communication with domain experts allowed us to grasp the business domain in no time and to implement its business logic. Yes, it was a big scary monolith, but for two developers in a garage it was just good enough. Again, we produced working software in a very aggressive time to market.

Domain-Driven Design

Our understanding of Domain-Driven Design at this stage could be represented with this simple diagram:



Ubiquitous language and an anemic domain model, in a monolithic bounded context.

Bounded Context #2: CRM

Soon after we deployed the campaign management solution, leads started flowing in, and we were in a rush. Our sales agents needed a robust CRM[61] system.

The CRM had to aggregate all incoming leads, group them based on different parameters, and distribute the leads across multiple sales desks around the globe. It also had to integrate with our clients' internal systems, both to notify the clients about changes in the leads' lifecycles and to complement our leads with additional information. And, of course, the CRM had to provide as many optimization opportunities as possible. For example, we needed the ability to make sure that the agents were working on the most promising leads, to assign leads to agents based on their qualifications and past performance, and to allow a very flexible solution for calculating agents' commissions.

Since no off-the-shelf product fit our requirements, we decided to roll out our own CRM system.

More Aggregates!

The initial implementation approach was the good ol' DDD Lite. We decided to call every noun an aggregate, and shoehorn them into the same monolith. This time, however, something felt wrong

[61] https://en.wikipedia.org/wiki/Customer-relationship_management

right from the start.

We noticed that, all too often, we were adding awkward prefixes to those "aggregate" names - like CRMLead and MarketingLead, MarketingCampaign and CRMCampaign. Interestingly, we never used those prefixes in conversations with domain experts — somehow, they always understood the meaning from the context. Then it dawned on me: there was a chapter with something about contexts in the Big Blue Book. This time, I read it cover to cover.

Achievement Unlocked: Read the Blue Book

I learned that bounded contexts solve exactly the same issue we had experienced — they protect the consistency of the ubiquitous language.

By that time, Vaughn Vernon had published his "Effective Aggregate Design"[62] paper. After reading it, I finally understood that aggregates aren't just data structures; they play a much larger role by protecting the consistency of the data.

We took a step back, and redesigned the CRM solution to reflect these revelations.

Solution Design: Take #2

We started by dividing our monolith into two distinct bounded contexts: Marketing and CRM. We didn't go all the way to microservices here, or anything like that. We just did the bare minimum to protect the ubiquitous language.

However, in the new bounded context — CRM — we were not going to repeat the same mistakes we did in the Marketing system. No more anemic domain models! Here we would implement a real domain model with real, by-the-book aggregates. In particular, we vowed that:

- Each transaction would affect only one instance of an aggregate;
- Instead of an OR/M, each aggregate itself would define the transactional scope; and
- The service layer would go on a very strict diet, and all the business logic would be refactored into the corresponding aggregates.

We were so enthusiastic about doing things the right way – but, soon enough, it became apparent that modeling a proper domain model is damn hard!

First Blood

Relative to the Marketing system, everything took much more time! It was almost impossible to get the transactional boundaries right the first time. We had to evaluate at least a few models and test them, only to figure out later that the one we hadn't thought about was the correct one. The price of doing things the "right" way was very high — lots of time.

[62] http://dddcommunity.org/library/vernon_2011/

Soon it became obvious to everyone that there was no chance in hell we would meet the deadlines! To help us out, the management decided to offload implementation of some of the features to … DBAs.

Yes, stored procedures.

This single decision resulted in much damage down the line. Not because SQL is not the best language for describing business logic. No, the real issue was a bit more subtle and fundamental.

Babel Tower 2.0

This situation produced an implicit bounded context whose boundary dissected one of our most complex business entities: the Lead.

The result was two teams working on the same business component and implementing closely related features, but with minimal interaction between them. Ubiquitous language? Give me a break! Literally, each team had its own vocabulary to describe the business domain and its rules.

Achievement Unlocked: Slapped by Conway's Law

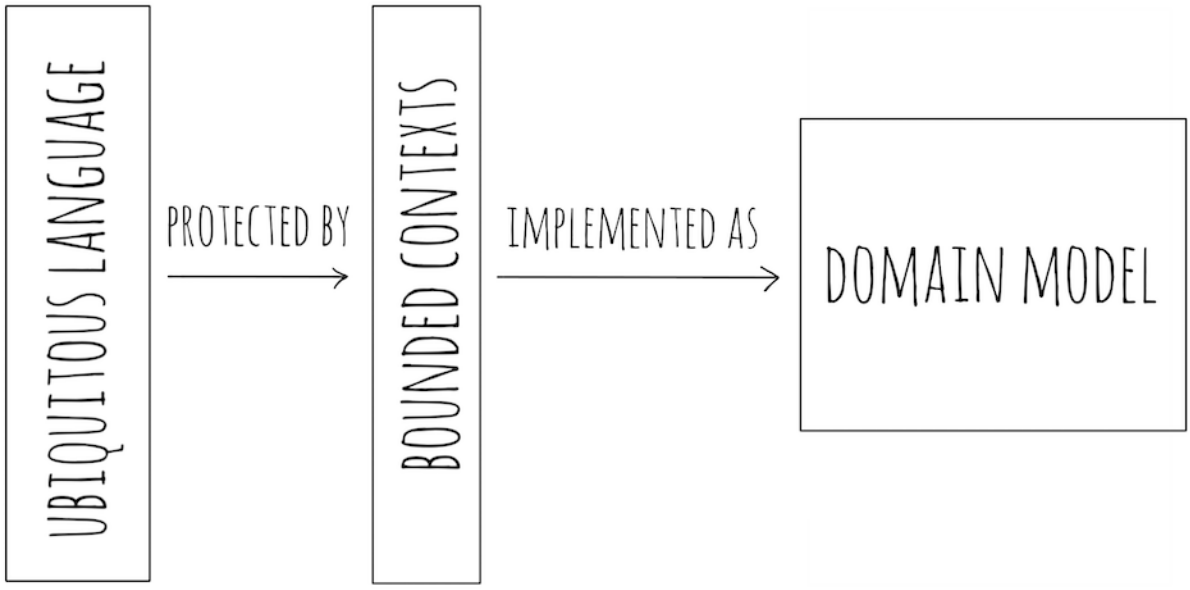
The models were inconsistent. The was no shared understanding. Knowledge was duplicated, the same rules were implemented twice. Rest assured, when the logic had to change, the implementations went out of sync immediately.

Needless to say, the project wasn't delivered anywhere near on time, and it was full of bugs. Nasty production issues that had flown under the radar for years corrupted our most precious asset — our data.

The only way out of this mess was to completely rewrite the Lead aggregate, this time with proper boundaries, which we did a couple of years later. It wasn't easy, but the mess was so bad there was no other way around it.

Domain-Driven Design

Even though this project failed pretty miserably by business standards, our understanding of Domain-Driven Design evolved a bit: build a ubiquitous language, protect its integrity using bounded contexts, and instead of implementing an anemic domain model everywhere, implement a proper domain model everywhere.



A Missing Piece?

Of course, a crucial part of Domain-Driven Design was missing here: subdomains, their types, and how they affect a system's design.

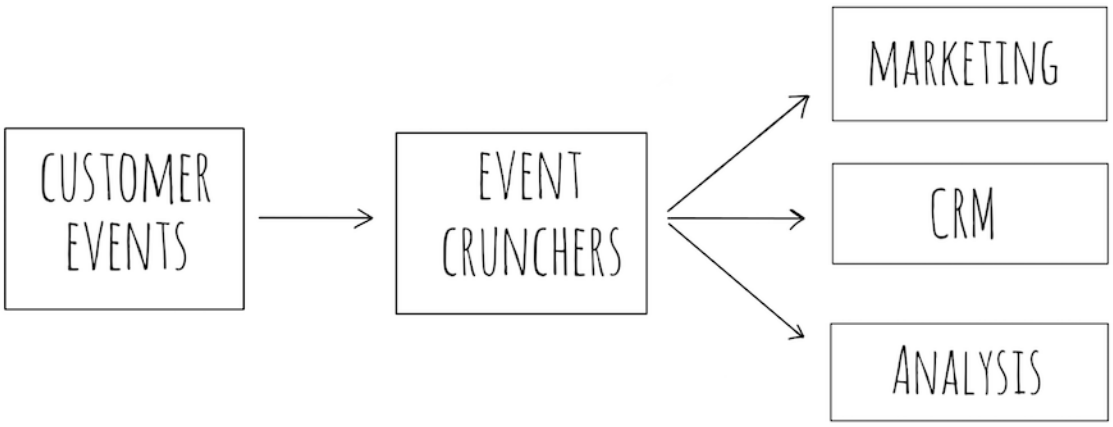
Initially we wanted to do the best job possible, but we ended up wasting time and effort on building domain models for supporting subdomains. As Eric Evans put it, not all of a large system will be well designed. We learned it the hard way, and wanted to use the acquired knowledge in the next project.

Bounded Context #3: Event Crunchers

After the CRM system was rolled out, we suspected that an implicit subdomain was spread across Marketing and CRM.

Whenever the process of handling incoming customer events had to be modified, we had to introduce changes both in the Marketing and CRM bounded contexts.

Since conceptually this process didn't belong to any of them, we decided to extract this logic into a dedicated bounded context called Event Crunchers.



Solution Design

Since we didn't make any money out of the way we move data around, and there weren't any off-the-shelf solutions that could have been used, Event Crunchers resembled a supporting subdomain. We designed it as such.

Nothing fancy this time: just layered architecture and some simple ETL-like transaction scripts. This solution worked great, but only for a while.

The Ball Started Rolling

As our business evolved, we implemented more and more features in the Event Crunchers. It started by BI people asking for some flags – a flag to mark a new contact, another one to mark various first-time events, some more flags to indicate some business invariants, etc.

Eventually those simple flags evolved into a real business logic, with complex rules and invariants. What started out as an ETL script evolved into a fully-fledged core business domain.

Unfortunately, nothing good happens when you implement complex business logic as ETL scripts. Since we didn't adapt our design to cope with the complex business logic, we ended up with a very big ball of mud. Each modification to the codebase became more and more expensive, quality went downhill, and we were forced to rethink the Event Crunchers design. We did it a year later.

Design: Take #2

By that time, the business logic became so complex that it could only be tackled with Event Sourcing. We refactored Event Crunchers' logic into an event-sourced domain model, with other bounded contexts subscribing to its events.

Interestingly, we had both similar and different experiences in another project.

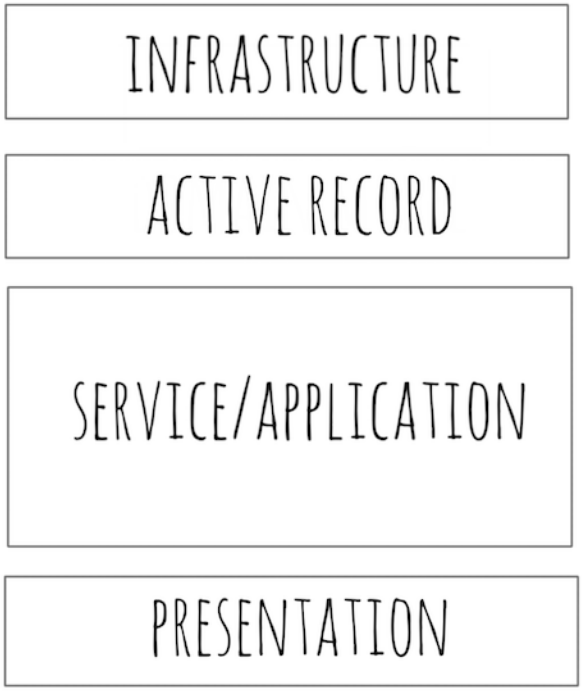
Bounded Context #4: Bonuses

One day, the sales desk managers asked us to automate a simple, yet tedious procedure that they had been doing manually: calculating the commissions for the sales agents.

Implementation

Again, it started out simply - once a month, just calculate a percentage of each agent's sales, and send the report to the managers. As before, we contemplated whether his was our core business domain. The answer was no. We weren't inventing anything new, weren't making money out of this process, and if it was possible to buy an existing implementation, we definitely would. Not core, not generic, but another supporting subdomain.

We designed the solution accordingly: some Active Record objects, orchestrated by a "smart" service layer:



Once the process became automated, boy, did everyone become creative about it.

Creativity Unleashed

Our analysts wanted to optimize the heck out of this process. They wanted to try out different percentages, to tie percentages to sales amounts and prices, to unlock additional commissions for achieving different goals, etc., etc. Guess when the initial design broke down?

Again, the codebase started turning into an unmanageable ball of mud. Adding new features became more and more expensive, bugs started to appear – and when you're dealing with money, even the smallest bugs can have BIG consequences.

Design: Take #2

As with the Event Crunchers project, at some point we couldn't bear it anymore. We had to throw away the old code and rewrite the solution from ground up, this time as an event-sourced domain model.

Just as in Event Cruncher, the business domain was initially categorized as a supporting one. As the system evolved, it gradually mutated into a core business domain: we found ways to make money out of these processes. However, there is a striking difference between these two bounded contexts.

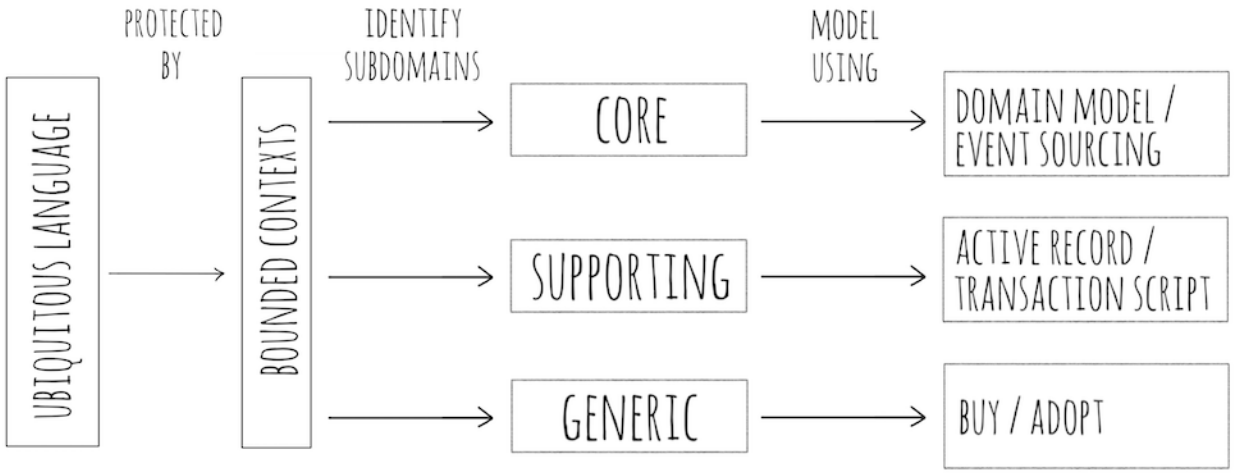
"Same Same, But Different"

For the Bonuses project, we had a ubiquitous language. Even though the initial implementation as based on Active Records, we could still have a ubiquitous language.

As the domain's complexity grew, the language used by the domain experts got more and more complicated as well. At some point, it could no longer be modeled using Active Records! This realization allowed us to notice the need for a change in the design much earlier than in the case of the Event Crunchers. We saved a lot of time and effort by not trying to fit a square peg in a round hole, thanks to the ubiquitous language.

Domain-Driven Design

At this point, our understanding of Domain-Driven Design had finally evolved into a classic one: ubiquitous language, bounded contexts, and different types of subdomains, each designed according to its needs.



However, things took quite an unexpected turn for our next project.

Bounded Context #5: The Marketing Hub

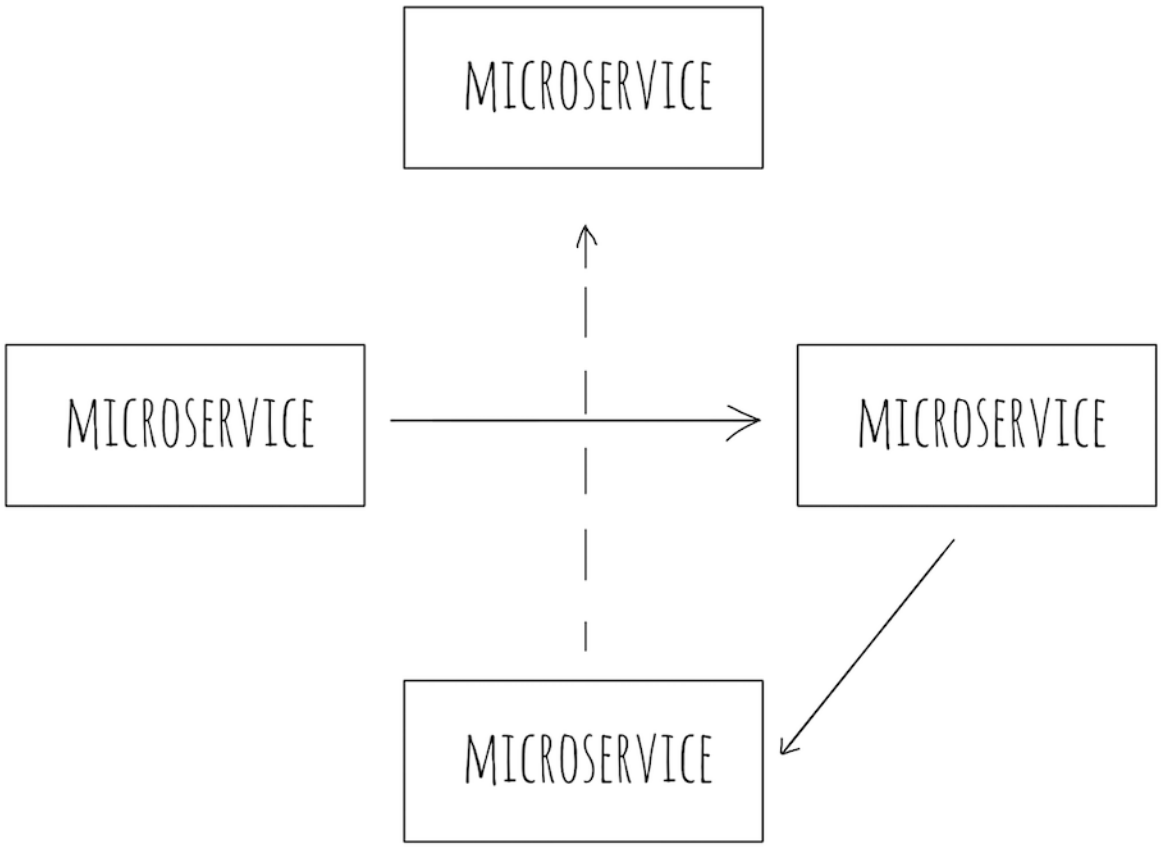
Our management was looking for a profitable new vertical. They decided to try using our ability to generate a massive number of leads, and sell them to smaller clients, ones we hadn't worked with before.

This project was called "Marketing Hub".

Design

Since this business domain was defined as a new profit opportunity by management, it was clearly a core business domain. Hence, design-wise, we pulled out the heavy artillery: Event Sourcing and CQRS. Also, back then, a new buzzword — microservices — started gaining lots of traction. We decided to give it a try.

This is what our solution looked like:



Small services, each having its own database, with both synchronous and asynchronous communication between them. On paper, it looked like a work of art. In practice, not so much.

Micro-what?

We naively approached microservices thinking that the smaller the service, the better. So we drew service boundaries around the aggregates. In DDD-lingo, each aggregate became a bounded context on its own.

Again, initially this design looked great. It allowed us to implement each service according to its specific needs. Only one would be using Event Sourcing, and the rest would be state-based aggregates. Moreover, all of them could be maintained and evolved independently.

However, as the system grew, those services became more and more chatty. Eventually, almost each service required data from all the other services to complete some of its operations. The result? What was intended to be a decoupled system ended up being a distributed monolith. And an absolute nightmare to maintain.

Unfortunately, there was another, much more fundamental issue we had with this architecture. To implement the Marketing Hub, we had used the most complex patterns for modeling the business domain: Event Sourcing and Domain Model. We carefully crafted all those services. But it all was in vain.

The Real Problem

Despite the fact that Marketing Hub was considered a core domain by the business, it had no technical complexity in it. Behind that complex architecture stood a very simple business logic – so simple that it could have been implemented using plain active records.

As it turned out, the business people were looking to profit by leveraging our existing relationships with other companies, and not through the use of clever algorithms.

The technical complexity ended up being much higher than the business complexity. To describe such discrepancies in complexities, we use the term “accidental complexity”, and our initial design ended up being exactly that. The system was over-engineered.

Bounded Contexts: Summary

Those were the five bounded contexts that I wanted to tell you about: Marketing, CRM, Event Crunchers, Bonuses, and Marketing Hub.

I know, it might sound like we’re a bunch of losers who can’t do anything right the first time. But fear not: I wanted to share the stories of the bounded contexts we learned the most from. And that brings us to the second part. Let’s see what we learned from this experience.

Part II: What We Learned

Ubiquitous Language

In my opinion, ubiquitous language is the "core domain" of Domain-Driven Design. The ability to speak the same language with our domain experts has been indispensable to us. It turned out to be a much more effective way to share knowledge than tests, documents, and "even" Jira.

Moreover, the presence of a ubiquitous language has been a major predictor of a project's success for us:

- When we started, our implementation of the Marketing system was far from perfect. However, the robust ubiquitous language compensated for the architectural shortcomings and allowed us to deliver the project's goals.
- In the CRM context, we screwed it up. Unintentionally, we had two languages describing the same business domain. We strived to have a proper design, but because of the communication issues we ended up with a huge mess.
- The Event Crunchers project started as a simple supporting subdomain, and we didn't invest in the ubiquitous language. We regretted this decision big time when the complexity started growing. It would have taken us much less time if we initially started with a ubiquitous language.
- In the Bonuses project, the business logic became more complex by orders of magnitude, but the ubiquitous language allowed us to notice the need for a change in the implementation strategy much earlier.

Hence, our take on it right now: ubiquitous language is not optional, regardless of whether you're working on a core, supporting, or even a generic subdomain.

Invest Early!

We also learned the importance of investing in the ubiquitous language as early as possible. It is practically impossible to "fix" a language if it has been spoken for a while in a company (as was the case with our CRM system). We were able to fix the implementation. It wasn't easy, but eventually we did it. That's not the case, however, for the language. To this day, some people still use the conflicting terms defined in the initial implementation.

Subdomains

We all know that, according to Domain-Driven Design, there are three types of business domains:

Core Subdomains

Things the company is doing differently from its competitors to gain competitive advantage. In our case, we came up with our unique way to optimize the lifecycle of advertising campaigns. We also built our own CRM solution to make sure that we could measure and optimize each step in the process of handling incoming leads.

As we saw earlier, both Bonuses and Event Crunchers became our core subdomains, since we found ways of using those systems to gain additional profits.

Supporting Subdomains

The things the company is doing differently from its competitors, but that do not provide any competitive advantage.

A good example of a supporting subdomain in our company is the Creative Catalog. We implemented our own solution for managing creative materials – not because storing those files in a particular way made us more profitable, but because we had to – without it we couldn't deliver our campaign management and optimization solution.

Generic Subdomains

Generic subdomains are all the things that all companies do in the same way. Those are the solved problems. For example, like all companies, we had to manage our systems' users. We could have implemented our own solution. It would definitely have taken us a long time and loads of effort. I'm sure that, security-wise, our solution wouldn't have been ideal. And even if it was 100% secure, it still wouldn't affect the company's profits in any way. That's why it's preferable to use a common, battle-proven solution.

Tactical Design

It is a common practice to use this categorization in business domains to drive design decisions:

- For the core subdomains, use the heavy artillery: Domain-Driven Design's tactical patterns, or Event Sourcing;
- Supporting subdomains can be implemented with a rapid application development framework; and
- Generic subdomains, in almost all cases, are cheaper and safer to buy or adopt than to implement yourself.

However, this decision model didn't work well for us.

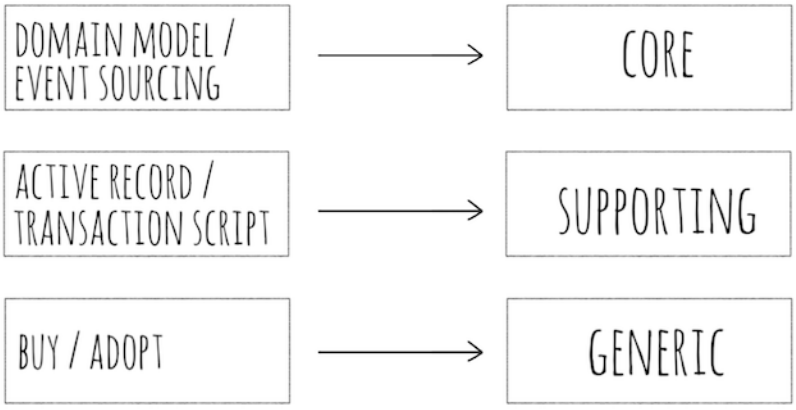
Companies, and especially startups like ours, tend to change and reinvent themselves over time. Businesses evolve, new profit sources are evaluated, other neglected, and sometimes unexpected opportunities are discovered. Consequently, business domain types change accordingly.

Speaking of our company, we have experienced almost all the possible combinations of such changes:

- Both the Event Crunchers and Bonuses started as *supporting* subdomains, but once we discovered ways to monetize these processes, they became our *core* subdomains.
- In the Marketing context, we implemented our own Creative Catalog. Nothing really special or complex about it. However, a few years later, an open-source project came out that offered even more features than we originally had. Once we replaced our implementation with this product, a *supporting* subdomain became a *generic* one.
- In the CRM context, we had an algorithm that identified the most promising leads. We refined it over time and tried different implementations, but eventually it was replaced with a fully managed machine learning model running on AWS. Technically, a *core* subdomain became *generic*.
- As we've seen, our Marketing Hub system started as a *core*, but ended up being a *supporting* subdomain, since the competitive edge resided in a completely different dimension.
- We also have quite a few examples in our industry of companies that turned *generic* and *supporting* subdomains into their *core* business. For example, Amazon and their AWS Cloud. Once this kind of change in a subdomain type happens, its design should evolve accordingly. Failing to do so in time will lead to blood, tears, and accidental complexities. Hence, instead of making design decisions based on subdomain types, we prefer to reverse this relationship.

From Tactical Design to Subdomains

For each subdomain, we start by choosing the implementation strategy first. No gold-plating here: we want the simplest design that will do the job.



Next, from the selected design, we deduce the subdomain's type. This approach has multiple benefits.

Benefit #1: Less Waste

The implementation is driven by the requirements at hand. It's not going to be over-engineered, as in the case of the Marketing Hub, and it won't be under-engineered, as in the case of the Bonuses project.

Benefit #2: Dialog

Second, reversing this relationship creates additional dialog between you and the business. Sometimes, business people need us as much as we need them.

If they think something is a core business, but you can hack it in a day, then questions should be raised about the viability of that business.

On the other hand, things get interesting if a subdomain is considered as a supporting one by the business but can only be implemented using advanced modeling techniques:

- First, the business people may have gotten over-creative with their requirements and ended up with accidental business complexity. It happens. In such a case, the requirements can, and probably should, be simplified.
- Second, it might be that the business people don't yet realize that they employ this subdomain to gain an additional competitive edge. (This happened in the case of the Bonuses project.) By uncovering this mismatch, you're helping the business to identify new profit sources much faster.

But how do you choose the implementation strategy? At Internovus, we've identified a couple of very simple heuristics that allow us to streamline this decision-making process.

Tactical Heuristics

To choose an implementation strategy for a business subdomain, you have to decide how to model its business logic in code. For that we have four options, four patterns:

Transaction Script

This pattern calls for implementing the business logic as a simple and straightforward procedural script. Nothing fancy or complicated. All you have to take care of is to make sure each operation is transactional: it either succeeds or fails. Hence the name — Transaction Script.

Active Record

Here we still have the same procedural code; however, instead of accessing databases directly, it operates on objects that abstract the persistence mechanism - active records.

Domain Model

I will use Domain Model as an overarching name for Domain-Driven Design's tactical patterns, such as aggregate, value object, service, etc.

Event Sourced Domain Model

This pattern is based on the domain model, but here we are using event sourcing to represent changes in aggregates' lifecycles.

Heuristics

You can decide which modeling pattern you should use by answering a number of questions about the business domain:

- Does the domain in question deal with money directly, require deep analytics or an audit log? If it does, use the event-sourced domain model.
- How complex is the business logic? Is it more complex than some input validations? Does it have complicated business rules and invariants? If it does, use the domain model.
- If the business logic is simple, then how complex are the data structures? If it includes complicated object relations or trees, implement the active record pattern.
- Lastly, if the answers to all those questions are negative, use a simple transaction script.

Architectural Patterns

Once you've decided how to model the business domain, mapping an architectural pattern is trivial:

- You need CQRS to implement event-sourced domain model.
- Domain model requires hexagonal architecture.
- Use layered architecture for active record.
- For transaction script, in many cases, you can even do without layers.

The only exception here is CQRS, as it can be useful for any of these patterns. But that's the next topic we're going to discuss.

Let's say you've chosen an implementation strategy, but over time it started breaking under its own weight. For example, you've been using the active record pattern, but maintaining the business logic became painful. This "pain" is a very important signal. Use it!

Don't Ignore Pain

It means that the business domain has evolved. It's time to go back and rethink its type and implementation strategy. If the type has changed, talk with the domain experts to understand the business context. If you do need to redesign the implementation to meet new business realities, don't be afraid of this kind of change. Once the decision of how to model the business logic is made consciously and you're aware of all the possible options, it becomes much easier to react to such a change, and to refactor the implementation to a more elaborate pattern.

Let's talk a bit more in-depth about CQRS.

Lesson #4: CQRS

Historically, CQRS is closely related to event sourcing. If you're doing event sourcing, in almost all cases you need CQRS. However, it is crucial to understand that event sourcing and CQRS are not the same. Instead, they are two conceptually different patterns.

CQRS is not Event Sourcing

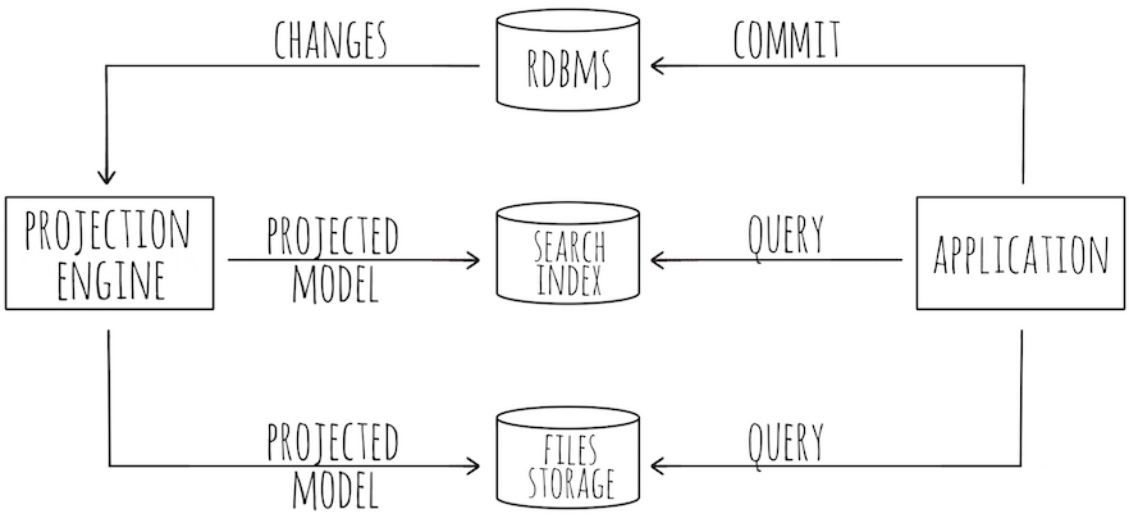
Event sourcing is a way to model the behavior of a business domain. CQRS, on the other hand, is an architectural pattern. It allows you to represent the same data in different persistent models. For example, in case of event sourcing, you have the event store for writing and projections for reading — the same data, but persisted in different models.

We found that CQRS can be very beneficial for all kinds of business domains, even those implemented as simple transaction scripts or active records. In such cases, we built the projections out of the persisted state. We called them state-based projections.

State Based Projections

The idea is the same as materialized views in relational databases, the difference being that we project additional models into other types of databases.

For example, in the Creative Catalog we used a relation model to represent the creatives. However, the data was projected into a search index(Elasticsearch) and plain files in S3 for caching:



What makes this implementation of CQRS, and not a mere data replication, is the fact that we used the same infrastructure to project additional models. It always allowed us to plug in new types of projections in the feature or to wipe and rebuild existing ones.

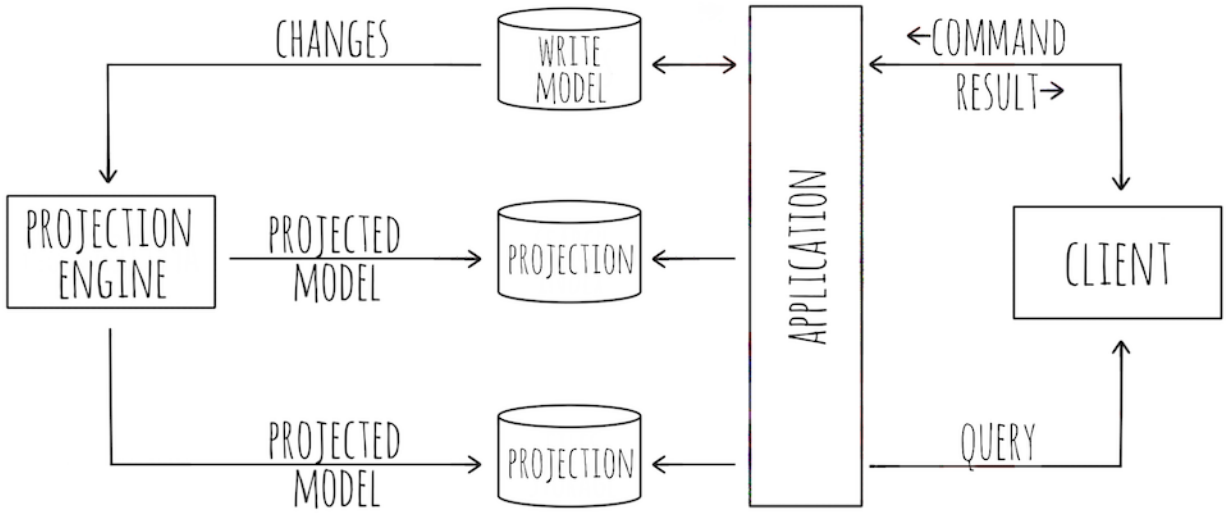
Relaxed Segregation

We all know that the CQRS pattern originated from the CQS principle. However, in our experience, clinging too tightly to the CQS roots resulted in nothing but accidental complexity. Let me explain.

Originally, we tried to keep our commands "void", i.e. not returning any data. The more we did it, the more we struggled, until finally we realized that it made little sense. When a user or a system executes a command, they need to know its outcome:

- Whether the command succeeded or failed.
- If it failed — why? Were there any validation issues? Or was there a technical issue?
- If it succeeded, the user experience may be improved by reflecting the updated data back in the UI.

We didn't find any reasons that prevented us from returning this information as a command's result. We did, however, segregate the models: since projections are eventually consistent, all the data returned by the command should originate from the strongly consistent "write" model:



Bounded Contexts' Boundaries

At Internovus, we've tried quite a few strategies for setting the boundaries of bounded contexts:

- Linguistic boundaries: We split our initial monolith into Marketing and CRM contexts to protect their ubiquitous languages;
- Subdomain-based boundaries: Most of our subdomains were implemented in their own bounded contexts – for example, Event Crunchers and Bonuses.

- Entity-based boundaries: As we discussed earlier, this approach had limited success in the Marketing Hub project, but it worked well in others. For example, later on we extracted some of the entities out of Marketing into their own bounded contexts.
- Suicidal boundaries: As you may remember, in the initial implementation of the CRM we dissected an aggregate into two different bounded contexts. Don't ever try this at home, okay?

Which of these strategies is the recommended one? Neither one fits all cases. Let's see why.

Heuristics

As Udi Dahan pointed out, finding boundaries is really hard — there is no flowchart for that! This statement has profound implications. Since there is no flowchart, the only way to find the right boundaries is by doing some trial-and-error work yourself. Which means, by definition, there will be mistakes. There is no way around it! So let's acknowledge this and only make mistakes that are easy to fix, and try to avoid the fatal ones.

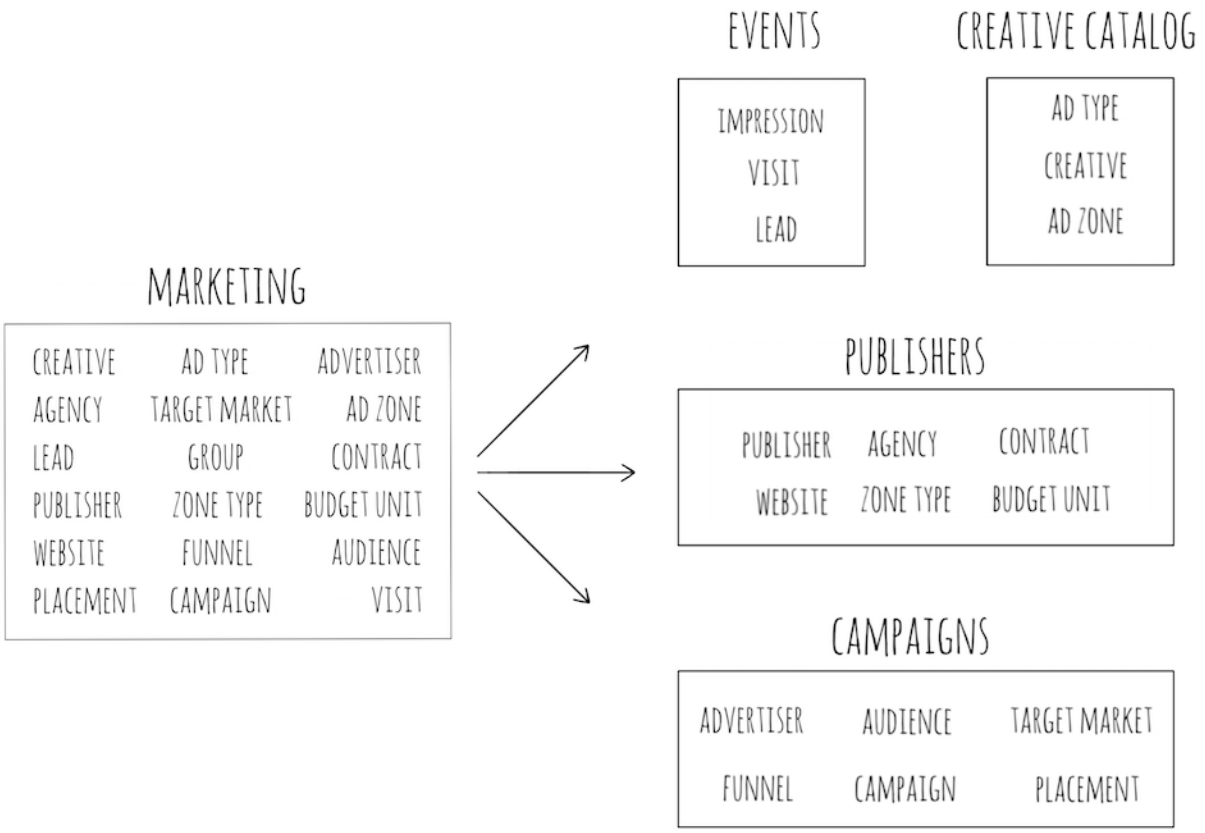
In our experience, it is much safer to extract a service out of a bigger one, than to start with services that are too small. Hence, we prefer to start with bigger boundaries and decompose them later, as more knowledge is acquired about the business. How wide are those initial boundaries? It all goes back to the business domain — the less you know about the business domain, the wider the initial boundaries.

- For more complex domains, start with bigger boundaries, e.g., linguistic or subdomain boundaries.
- Simpler ones, like supporting subdomains, can be decomposed earlier, e.g., into subdomain, or even entity-based boundaries.

However, make sure to extract those smaller contexts – not because you can, but only if you truly need them to deliver some functional or non-functional requirements.

Examples

When we tried to decompose to fine-grained services early on, as in the case of Marketing Hub, we ended up with a distributed monolith. On the contrary, the Marketing context started as a very wide one, but was decomposed later on; we extracted Campaigns, Publishers, Creative Catalog, and Events into separate bounded contexts:



Again, it is evident even here: the simpler the business domain, the narrower its boundaries.

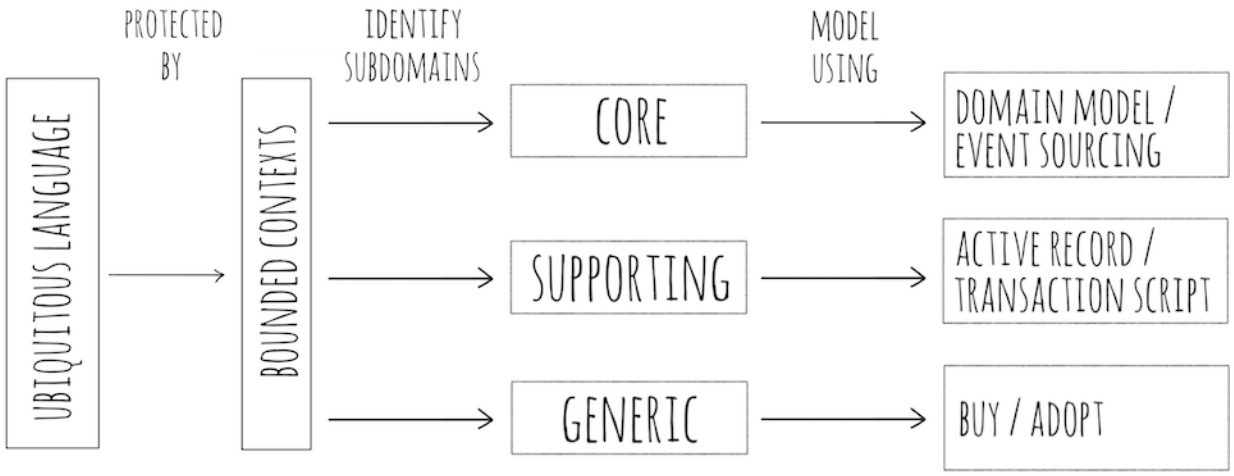
Summary

Those are the five pieces of practical advice I wanted to share:

- Ubiquitous language is not optional in any kind of business domain;
- Subdomain types change — so embrace those changes! Use them to make your design more resilient;
- Learn the ins and outs of the four business logic modeling patterns, and use them where each is appropriate;
- Use CQRS to represent data in multiple persistent models, and don't be too dogmatic about the segregation; and
- Start with wide bounded context boundaries, and decompose them further when more knowledge of the business domain is acquired — but only if you have good reasons to do so.

Domain-Driven Design

Our current way of applying Domain-Driven Design is as follows:



1. We always start by building a ubiquitous language with the domain experts, in order to learn as much as possible about the business domain.
2. In the case of conflicting models, we decompose the solution into bounded contexts, following the linguistic boundaries of the ubiquitous language.
3. We choose an implementation strategy for each subdomain by using the heuristics I've shown you.
4. From this design, we deduce the types of business domains at play and verify them with the business. Sometimes this dialog leads to changes in the requirements, because we are able to provide a new perspective on the project to the product owners.
5. As more domain knowledge is acquired, and if needed, we decompose the bounded contexts further into contexts with narrower boundaries.

The main difference between our current vision of Domain-Driven Design and the one we started with is that we went from "Aggregates Everywhere" to *"Ubiquitous Language Everywhere"*.

P.S.

Since I've told you the story of how Internovus started, I want share how it ended.

The company became profitable very quickly, and 7 years after its inception it was acquired by our biggest client. Of course, I cannot attribute its success solely to Domain-Driven Design. However, during those 7 years, we were constantly in "startup mode".

What we term “startup mode” in Israel, in the rest of the world is called “chaos”: constantly changing business requirements and priorities, aggressive timeframes, and a tiny R&D team. DDD allowed us to tackle all these complexities and keep delivering working software. Hence, when I look back, the bet we placed on Domain-Driven Design 7 years ago paid off in full.

Tackling Complexity in ERP Software: a Love Song to Bounded Contexts — Machiel de Graaf and Michiel Overeem

Enterprise resource planning (ERP) software offers an integrated solution for the management of core business processes. Different domains (accounting, sales, relation management, payrolling) are handled by a single software system. A well-executed ERP system has the benefit that information is only stored once, and relations between information are easy to follow since it is all in one system.

We, AFAS Software in the Netherlands, have two decades of experience with building ERP software, and not without success. After working on the same codebase for so long, we strategically decided to do a rewrite. Yes, we know that Joel Spolsky said not to do that[63]. We are in the DHH-camp[64], because the new version needs to be different. We needed to tackle the complexity in ERP software. Through our discovery of CQRS and event-sourcing we stumbled upon Domain-Driven Design (DDD). And the goal of DDD is to tackle the complexity in software, exactly what we needed. We let DDD inspire us as much as possible. But we also decided to take it a step further. We experienced three major challenges in developing ERP software that we wanted to tackle in this new version.

Disclaimer: although we have quite a journey behind us, we have not arrived yet. The software is not yet in production, and our experiences come solely from testing the system in simulated environments.

The three challenges we faced

The first challenge is that ERP software has **a lot of repetitive logic**. There are numerous parts that look very similar. For instance, the maintenance of a product catalog is not that different from the maintenance of a list of organizations, from a technical point of view. In a large software system, if you are not careful, the two will be developed in isolation, and will work inconsistent. This inconsistency will bother end-users, because they have certain expectations of the usability of the system. Evolution of one of the maintenance components will also require extra work, because similar parts of the system need to be kept consistent. Obviously, ERP systems gain a lot from re-use and standards.

Developers can benefit a lot from reuse in ERP systems: it can increase productivity and quality. However, reuse can become a pain in the long run. Changing the library that is used in numerous

[63] https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/
[64] https://signalvnoise.com/posts/3856-the-big-rewrite-revisited

places is hard and can result in unpredictable outcomes. New developers do not know the history of that library, and do not have the bigger picture to oversee their changes. Teams must be careful with reuse, while it looks temping in the beginning, refactorings can be very expensive.

The second challenge is that ERP software requires **a high level of customizability**. Processes within businesses look similar, but when customers are spread out over different branches and all have their specific needs, the processes are not the same. While it is possible to sell off-the-shelve ERP software, the software still needs options for customization through configuration and parameterization.

These configuration options result in additional complexity within the code. Every parameter adds branches through the control flow and requires additional tests and effort during code changes. The developer needs to understand all these different paths, and more important, be able to predict the influence of his change on all these paths.

The third challenge that we faced was **the mix of domain knowledge and technical knowledge in one code base**. While functionality has a certain evolution path (law, improvements, new branches), technology has a completely different pace (move from on-premise to cloud for instance). Not only the pace is different, it is also difficult to have expert level domain knowledge and have the technical skills to build and maintain a large system. While teams work closely together, we still end up with creating functional requirements and designs that a technical specialist translates into code.

We knew that if we would do a straight-forward rewrite of the ERP system, we would end up with the same mix a couple years down the line. What we needed was a clear separation of the two concerns that would allow us to evolve them in a different pace. And maybe even involve the business analysists more with the functionality expressed in code.

Making the domain a first-class citizen

We found a solution for our challenges in the utilization of model-driven development (MDD). Model-driven development is claimed to increase both productivity and quality, while lowering complexity by raising the level of abstraction. By creating a meta-model[65] of the domain, instances of that meta-model can be created to describe the software on a higher level. An automated process, or generator, then translates this model into running software.

The domain of the software itself is expressed in the meta-model and is thus pulled out of the actual source code. This allows us to bring our code re-use to a new level, because many instances of a single pattern can be translated into running software by a single transformation. It also enables a new level of variability: by changing the model the system changes. Variability is at the core of the system. Finally, it enables us to separate functionality and technology, and let them evolve at different rates.

The development process that we started based on this approach consists of four simple steps:

1. Extract patterns from the existing business logic and formalize those patterns in a meta-model.

[65] A meta-model in MDD is like a domain specific language, but does not need a concrete syntax.

2. Use that meta-model to create a model of the business logic in an ERP system.
3. Generate a working application from that model.
4. Evaluate the working application, and when necessary start at 1.

These steps might appear simple, however, executing them is not that easy. And while MDD is a large research field with promising results, it also has its share of failed projects. However, we believe there are two important reasons why we can pull this off.

First, we are in the ERP business for almost 20 years. We know our domain. We have built it, and we are shipping ERP software as we speak. We have a large group of experts in our company, and we have an even larger group of customers that show us how they use our software and what they need.

Second, we are not trying to build an MDD platform that allows you to build a large variety of software systems. The platform is only used to build our own ERP system, the platform itself is not the product, it is a tool for us to develop our product. We know the boundaries and scope of our domain. We do not have to solve every problem. We only need to solve our own, ERP related, problems.

One bounded context

As mentioned, we discovered DDD through CQRS and event-sourcing. And we choose to generate a CQRS, event-sourced system from the model, because we believed in the scale and flexibility it would provide us. However, in the initial versions of our platform we still had two monoliths: our generator, and the generated CQRS application. One of the biggest inspirations that we have found in DDD was the concept of bounded contexts, of different sub-domains, and the idea that the DRY (don't repeat yourself) rule should only be used within a bounded context[66].

The resulting application was a monolithic deployment unit. With every change of the model, we needed to generate the complete application, and deploy it. Now, it isn't impossible to do (we are doing it, and even manage to deploy a new application with zero-downtime through blue-green deployments), but the deployment feels bigger and more cumbersome than necessary. We feel that this kind of upgrades will not scale and that it will become a problem later one, when the model starts evolving at an increased speed.

We also had a single monolithic generator that translated the complete model. This became a burden when both the meta-model (the number of possible patterns increased) and the development team grew. The monolithic generator had several generic possibilities, and new features needed to be solved in the same generic way. This let to sub-optimal solutions. It also made the generator very complex. There were so many possible execution paths that no-one was able to reason about them. We all lost the big picture of the generator. We tried to solve these problems by

[66] https://medium.com/russmiles/on-boundaries-and-microservices-d559ec52bb55

1. introducing abstraction (*as we all know; every problem can be solved by adding more indirection*[67]).
2. refactoring our generator into a multi-stage generator. The parsed model was transformed, extended, simplified, translated into intermediate models, before generating the final output.

In the long run this did not work out. We still had a single transformation pipeline on which the whole team was working. The development work was not scaling, because we were getting in the way of each other. There was no real code ownership, so no team felt responsible. And finally, it made it hard to implement an incremental generation flow.

With the huge amount of code that we were generating, we knew we had a new challenge to face. Two major steps took us from these two single bounded contexts (the generator and the resulting application) into a new realm of multiple bounded contexts:

1. The move from a single code generator into a collection of small generator *modules*.
2. The move from a monolithic runtime application into a platform and many loosely coupled *bundles*.

A swarm of generators

Generating an application involves a lot of steps. The input model is analysed, and patterns are translated into concepts that fit the target platform. While we were extending the meta-model and handling new target components, our generator became a big ball of mud.

As already mentioned, we tried to mitigate this by adding abstractions and by added different phases in the generator. But we were not able to untangle our big ball of mud. The solution was found in the concept of different bounded contexts.

We started to hard work of breaking up our monolithic generator into several smaller generators: *modules*. A module is in fact a mini-generator, responsible for specific model elements in the meta-model. One module might transform the elements of type x, while another transforms the elements of type y. How many instances of a pattern are present in a model is not important, the module will translate all occurrences into code that matches the target platform.

We made the different patterns in the meta-model the bounded contexts of our generator. A module is an independent component in the generator. It receives the model as input and delivers output. How a module works is a matter of implementation detail. It could be through model transformations and intermediate models, but it could also be in a single step. A module truly forms a bounded context in the generator: it uses its own language and makes its own decisions.

Every team is responsible for a number of these modules. There is clear ownership and responsibility, but there is also freedom. This allows our teams to move fast, without being hindered by the complexity of other parts of the generator.

[67] "Any problem in computer science can be solved with another level of indirection." is attributed to David Wheeler by **Diomidis Spinellis**. Another level of indirection. In Andy Oram and Greg Wilson, editors, Beautiful Code: Leading Programmers Explain How They Think, chapter 17, pages 279–291. O'Reilly and Associates, Sebastopol, CA, 2007.

Of course, this does introduce code duplication: several modules might need to make the same analysis or transformation. If these modules are developed by the same team, they can easily choose for reuse. However, it becomes more complex when these modules are developed by different teams. These scenarios are dealt with on a case-per-case basis. Some are solved with shared libraries, while some are not solved at all.

Breaking up the monolith in bundles

As explained, our runtime application architecture is based on CQRS and event-sourcing. We generated components such as aggregates, commands, events, and projectors from the input model. Still, our application was a distributed monolith in terms of deployment and upgradability.

When we started out, we generated C# and packaged everything in DLL's. And because our generator was monolithic, these DLL's contained all kinds of different functionality. A change to some small part meant that we needed to upgrade several DLL's at once.

Our second move started with the recognition of bounded contexts in our resulting application. Of course, there are many different parts in an ERP system, and our goal was the ability to update them individually. We needed to separate them, and we did that by no longer generating C#.

The process flow in any MDD platform is quite simple, when zoomed out: parse the model, analyse it and transform it into an executable form. That is the 'code generation' flow. When using interpretation, the process flow is a bit different. The executable is already there, and it parses the model, analyses it, and executes it. The main difference between code generation and interpretation is thus the moment in time when the model is parsed, analysed and executed. With code generation, the model is parsed and analysed well before any end-user touches the system. With interpretation, the model is parsed and analysed around the same time that the end-user touches the system.

We decided to move towards a hybrid form: we generate an intermediate language (in our case consisting of JSON files) that is interpreted at run-time. This intermediate language enables us to deploy smaller parts of the application, parts that we call *bundles*. The idea behind a bundle is that it is some isolated piece of functionality that should be deployable and upgradeable on its own. Technically, they're just some JSON files. You could see them as microservices, but we prefer to call them bundles.

A nice benefit of the move towards these definition files that are interpreted is that it allows us to upgrade the application without restarting the process. A .NET process cannot reload DLL's while running. It needs to be stopped, the DLL's need to be replaced, and then the process can be started. We can, however, update the JSON files, and the application can pick up the changes without being restarted. This gives us much more flexibility when it comes to upgrading.

It also allows us to create a multi-tenant host process that contains applications for different customers. The process can be shared between applications, giving us an improved resource usage. The underlying framework, with interpreters and other components, is what we are calling the 'platform'. It is our custom developed runtime platform that hosts the bundles and makes sure that everything works.

Final words

The move to modules and bundles has made us more agile and more flexible. Because of the recognition and application of bounded contexts in both the generated application and the generator we are in control again. Are there only upsides? No, as with every design decision there are trade-offs.

- We lose opportunities for reuse. While we explained that this is deliberate (only do DRY in a bounded context), it is still a discussion that we have every now and then. We try to mitigate this with small libraries, or by refactoring our bounded contexts. And in the long run: it is a deliberate choice we made.
- Debugging and analysing the JSON files at run-time is a lot harder than reading C# code.
- Upgrading the monolithic application with a blue-green strategy is far easier than upgrading several loosely coupled microservices that have some relation to each other. Our upgrade scenarios have become more complex.
- Finding the right bounded contexts is in fact hard work. Throwing everything together is easier than really thinking about boundaries and dependencies.
- We have lost safety in the generation process. The compilation of C# gave us a warning when different parts were not aligned. With the generation of JSON files we lose that warning.

Now of course, we do try to mitigate some of these challenges. We have, for instance, added explicit usage of contracts between bundles. Every bundle can provide and assume API's. The 'assumptions' are linked to the 'provides', and by doing that we get a graph of the bundles. The resulting graph forms the base for our consistency check and give us insight in the message flows in our application.

We are not yet at the end of our journey. There is road that needs to be walked, until we go live. And while walking that road we will let the great ideas in the DDD community inspire us!

Calm Your Spirit with Bounded Contexts – Julie Lerman

Discovering, honing and respecting boundaries is one of the most important facets of Domain-Driven Design. And the awareness of this practice is like a meditation that can be leveraged far beyond the efforts of building software. Dividing big, messy problems into smaller solvable problems can help one evolve towards the calmness that Eric Evans emanates. I have often said that Eric is like a philosopher - a deep, contemplative, big-idea thinker. Those of us who have had the opportunity to see him speak or speak with him, are fortunate to witness his mind at work as it wanders into various corners of an idea.

I often use Eric's analogy of cells and membranes to describe bounded contexts. But cells are also part of all living things and DDD itself is a living thing.

In its 15th year, at Explore DDD, Eric expressed his gratitude to the DDD community for understanding this - adding to it and evolving DDD. And he was earnest in his request that the community continue to evolve DDD, continue its life, the growth of new cells and - as needed - the death of cells, as well. DDD was already a gift, yet this blessing to participate in his work seems to be an even greater one.

There are many great minds in the DDD community and I'm excited to see where else DDD will take us. Even more, I look forward to watching Eric take pride as he witnesses and contributes to the future of DDD. And he will surely continue to encourage the community as we spread the power of DDD.

Personally, I'm so very grateful for the support that Eric and others in the DDD community have given to me as I share the DDD story in my own way with people who share my background and perspective.