Table of Contents

The Rationale for Domain-Driven Design

> The being cannot be termed rational or virtuous, who obeys any authority, but that of reason.
>
> — Mary Wollstonecraft

Introduction

According to the Project Management Institute's (PMI) *Pulse of the Profession* report published in February 2020, only **77%** of all projects meet their intended goals — and even this is true only in the most mature organizations. For less mature organizations, this number falls to just **56%** i.e. approximately one in every two projects does not meet its intended goals. Furthermore, approximately one in every five projects is declared an outright failure. At the same time, we also seem to be embarking on our most ambitious and complex projects.

In this chapter, we will examine the main causes for project failure and look at how applying domain-driven design provides a set of guidelines and techniques to improve the odds of success in our favor. While Eric Evans wrote his classic book on the subject way back in 2003, we look at why that work is still extremely relevant in today's times.

Why do software projects fail?

> Failure is simply the opportunity to begin again, this time more intelligently.
>
> — Henry Ford

According to the project success report published in the Project Management Journal of the PMI, the following six factors need to be true for a project to be deemed successful:

Table 1. Project Success Factors

Category	Criterion	Description
Project	Time	It meets the desired time schedules
	Cost	Its cost does not exceed budget
	Performance	It works as intended
Client	Use	Its intended clients use it
	Satisfaction	Its intended clients are happy
	Effectiveness	Its intended clients derive direct benefits through its implementation

With all of these criteria being applied to assess project success, a large percentage of projects fail for one reason or another. Let's examine some of the top reasons in more detail:

Inaccurate requirements

PMI's *Pulse of the Profession* report from 2017 highlights a very starking fact — a vast majority of projects fail due to inaccurate or misinterpreted requirements. It follows that it is impossible to build something that clients can use, are happy with and makes them more effective at their jobs if the wrong thing gets built — even much less for the project to be built on time, and under budget.

IT teams, especially in large organizations are staffed with mono-skilled roles such as UX designer, developer, tester, architect, business analyst, project manager, product owner, business sponsor, etc. In a lot of cases, these people are parts of distinct organization units/departments — each with its own set of priorities and motivations. To make matters even worse, the geographical separation between these people only keeps increasing. The need to keep costs down and the current COVID-19 ecosystem does not help matters either.

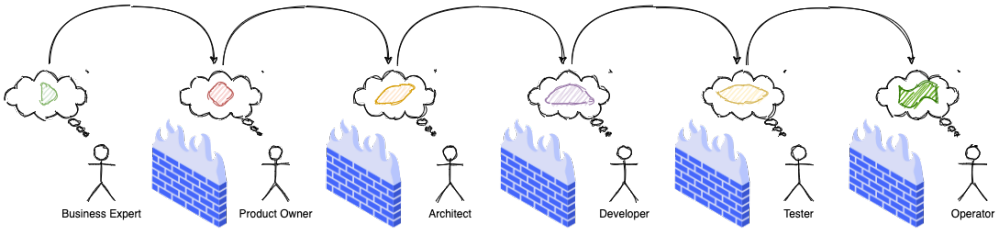


Figure 1- 1. Silo mentality and the loss of information fidelity

All this results in a loss in fidelity of information at every stage in the *assembly line*, which then results in misconceptions, inaccuracies, delays and eventually failure!

Too much architecture

Writing complex software is quite a task. One cannot just hope to sit down and start typing code — although that approach might work in some trivial cases. Before translating business ideas into working software, a thorough understanding of the problem at hand is necessary. For example, it is not possible (or at least extremely hard) to build credit card software without understanding how credit cards work in the first place. To communicate one's understanding of a problem, it is not uncommon to create software models of the problem, before writing code. This model or collection of models represents the understanding of the problem and the architecture of the solution.

Efforts to create a perfect model of the problem — one that is accurate in a very broad context, are not dissimilar to the proverbial holy grail quest. Those accountable to produce the architecture can get stuck in analysis paralysis and/or big design up front, producing artifacts that are one or more of too high level, wishful, gold-plated, buzzword-driven, disconnected from the real world — while not solving any real business problems. This kind of *lock-in* can be especially detrimental during the early phases of the project when knowledge levels of team members are still up and coming. Needless to say, projects adopting such approaches find it hard to meet with success consistently.

For a more comprehensive list of modeling anti-patterns, refer to Scott W. Ambler's website (http://agilemodeling.com) and book dedicated to the subject.

Too little architecture

Agile software delivery methods manifested themselves in the late 90s, early 2000s in response to heavyweight processes collectively known as *waterfall*. These processes seemed to favor big design up front and abstract ivory tower thinking based on wishful, ideal world scenarios. This was based on the premise that thinking things out well in advance ends up saving serious development headaches later on as the project progresses.

In contrast, agile methods seem to favor a much more nimble and iterative approach to software development with a high focus on working software over other artifacts such as documentation. Most teams these days claim to practice some form of iterative software development. However, this obsession to claim conformance to a specific family of agile methodologies as opposed to the underlying principles, a lot of teams misconstrue having just enough architecture with having no perceptible architecture. This results in a situation where adding new features or enhancing existing ones takes a lot longer than what it previously used to — which then accelerates the

devolution of the solution to become the dreaded big ball of mud.

Excessive incidental complexity

Mike Cohn popularized the notion of the test pyramid where he talks about how a large number of unit tests should form the foundation of a sound testing strategy — with numbers decreasing significantly as one moves up the pyramid. The rationale here is that as one moves up the pyramid, the cost of upkeep goes up copiously while speed of execution slows down manifold. In reality though, a lot of teams seem to adopt a strategy that is the exact opposite of this — known as the testing ice cream cone as depicted below:

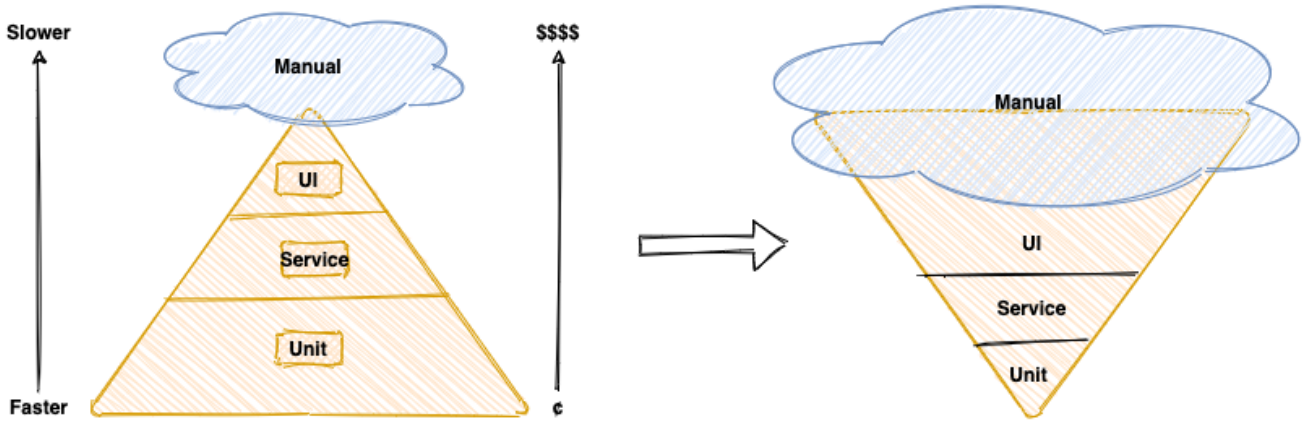


Figure 1- 2. Testing Strategy: Expectation vs. Reality

The testing ice cream cone is a classic case of what Fred Brooks calls incidental complexity in his seminal paper titled No Silver Bullet — Essence and Accident in Software Engineering. All software has some amount of essential complexity that is inherent to the problem being solved. This is especially true when creating solutions for non-trivial problems. However, incidental or accidental complexity is not directly attributable to the problem itself — but is caused by limitations of the people involved, their skill levels, the tools and/or abstractions being used. Not keeping tabs on incidental complexity causes teams to veer away from focusing on the real problems, solving which provide the most value. It naturally follows that such teams minimize their odds of success appreciably.

Uncontrolled technical debt

Financial debt is the act of borrowing money from an outside party to quickly finance the operations of a business — with the promise to repay the principal plus the agreed upon rate of interest in a timely manner. Under the right circumstances, this can accelerate the growth of a business considerably while allowing the owner to retain ownership, reduced taxes and lower interest rates. On the other hand, the inability to pay back this debt on time can adversely affect credit rating, result in higher interest rates, cash flow difficulties, and other restrictions.

Technical debt is what results when development teams take arguably sub-optimal actions to expedite the delivery of a set of features or projects. For a period of time, just like borrowed money allows you to do things sooner than you could otherwise, technical debt can result in short term speed. In the long term, however, software teams will have to dedicate a lot more time and effort towards simply managing complexity as opposed to thinking about producing architecturally sound solutions. This can result in a vicious negative cycle as illustrated in the diagram below:

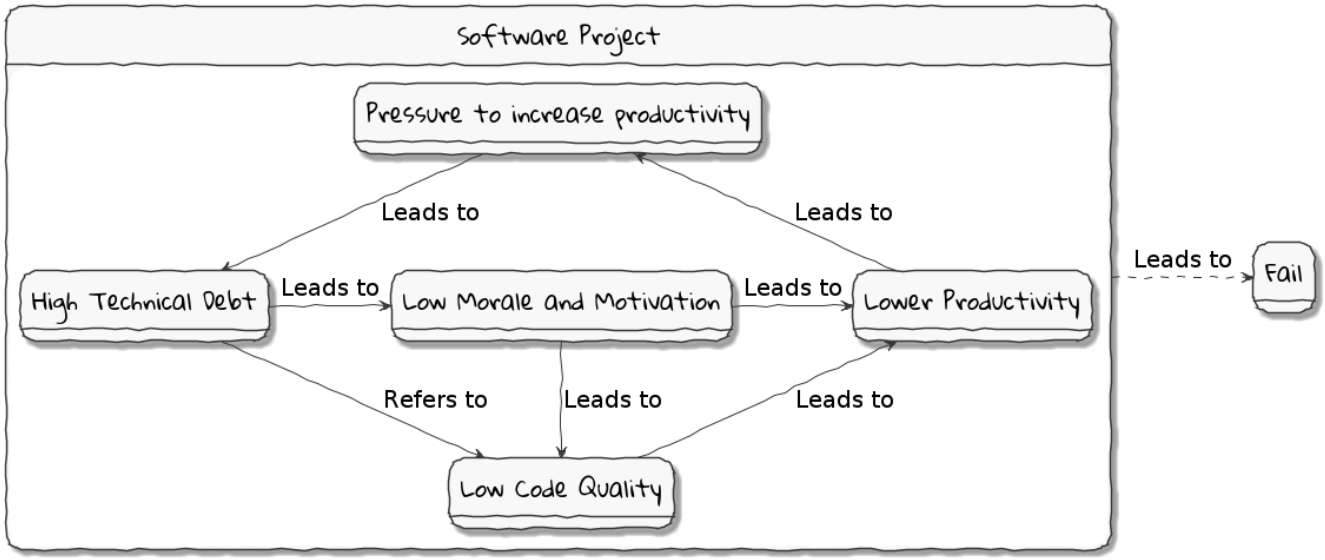


Figure 1- 3. Technical Debt — Implications

In a recent McKinsey survey sent out to CIOs, around 60% reported that the amount of tech debt increased over the past three years. At the same time, over 90% of CIOs allocated less than a fifth of their tech budget towards paying it off. Martin Fowler explores the deep correlation between high software quality (or the lack thereof) and the ability to enhance software predictably. While carrying a certain amount of tech debt is inevitable and part of doing business, not having a plan to systematically pay off this debt can have significantly detrimental effects on team productivity and ability to deliver value.

Ignoring Non-Functional Requirements (NFRs)

Stakeholders often want software teams to spend a majority (if not all) of their time working on features that provide enhanced functionality. This is understandable given that such features provide the highest ROI.These features are called functional requirements.

Non-functional requirements, on the other hand, are those aspects of the system that do not affect functionality directly, but have a profound effect on the efficacy of those using these using and maintaining these systems. There are many kinds of NFRs. A partial list of common NFRs is depicted below:

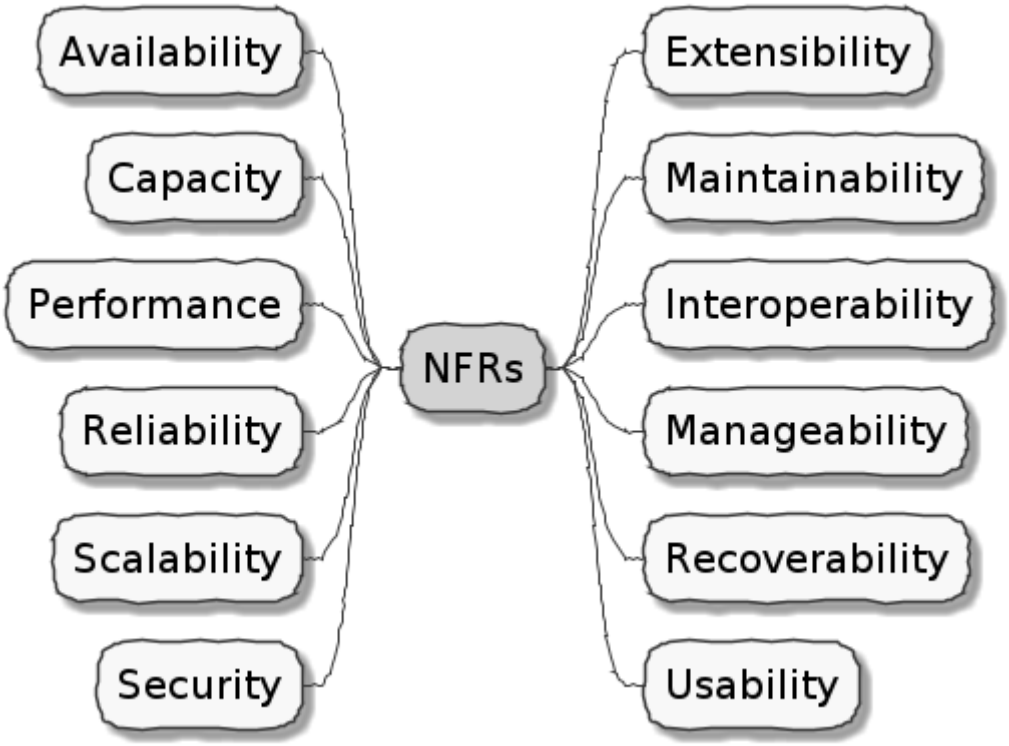


Figure 1- 4. Non-Functional Requirements

Very rarely do users explicitly request non-functional requirements, but almost always expect these features to be part of any system they use. Oftentimes, systems may continue to function without NFRs being met, but not without having an adverse impact on the *quality* of the user experience. For example, the home page of a web site that loads in under 1 second under low load and takes upwards of 30 seconds under higher loads may not be usable during those times of stress. Needless to say, not treating non-functional requirements with the same amount of rigor as explicit, value-adding functional features, can lead to unusable systems — and subsequently failure.

In this section we examined some common reasons that cause software projects to fail. Is it possible to improve our odds? Before we do that, let's look at the nature of modern software systems and how we can deal with the ensuing complexity.

Modern systems and dealing with complexity

> We can not solve our problems with the same level of thinking that created them.
>
> — Albert Einstein

As we have seen in the previous section, there are several reasons that cause software endeavors to fail. In this section, we will look to understand how software gets built, what the currently prevailing realities are and what adjustments we need to make in order to cope.

How software gets built

Building successful software is an iterative process of constantly refining knowledge and expressing it in the form of models. We have attempted to capture the essence of the process at a

high level here:

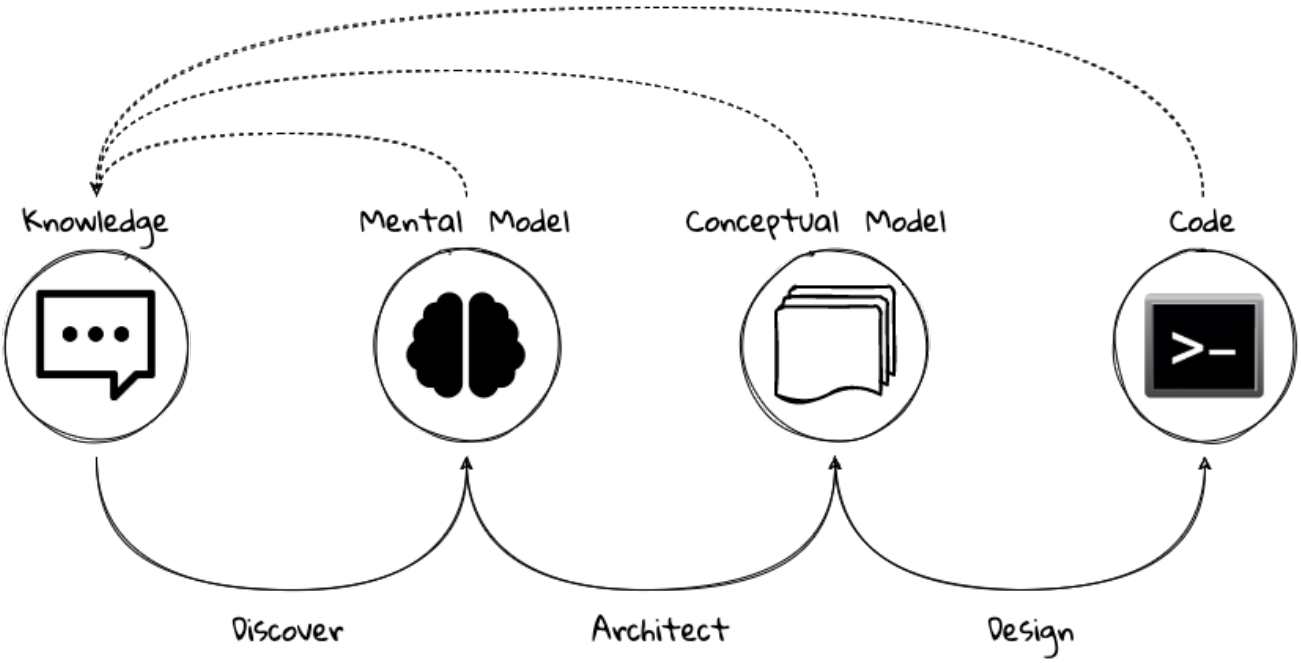


Figure 1- 5. Building software is a continuous refinement of knowledge and models

Before we express a solution in working code, it is necessary to understand **what** the problem entails, **why** the problem is important to solve, and finally, **how** it can be solved. Irrespective of the methodology used (waterfall, agile, and/or anything in between), the process of building software is one where we need to constantly use our knowledge to refine mental/conceptual models to be able to create valuable solutions.

Complexity is inevitable

We find ourselves in the midst of the fourth industrial revolution where the world is becoming more and more digital — with technology being a significant driver of value for businesses. Exponential advances in computing technology as illustrated by Moore's Law below,

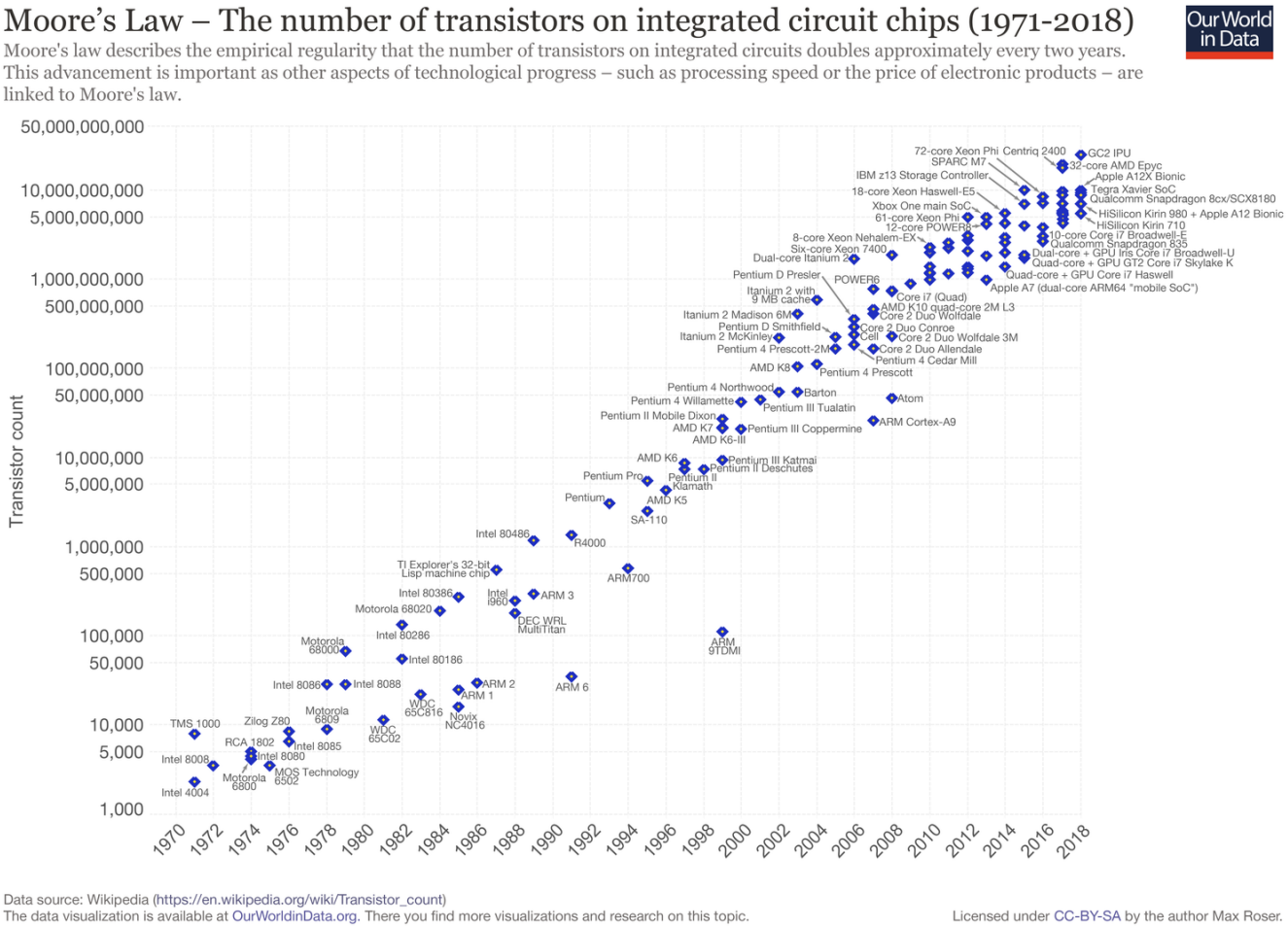


Figure 1- 6. Moore's Law

along with the rise of the internet as illustrated below,

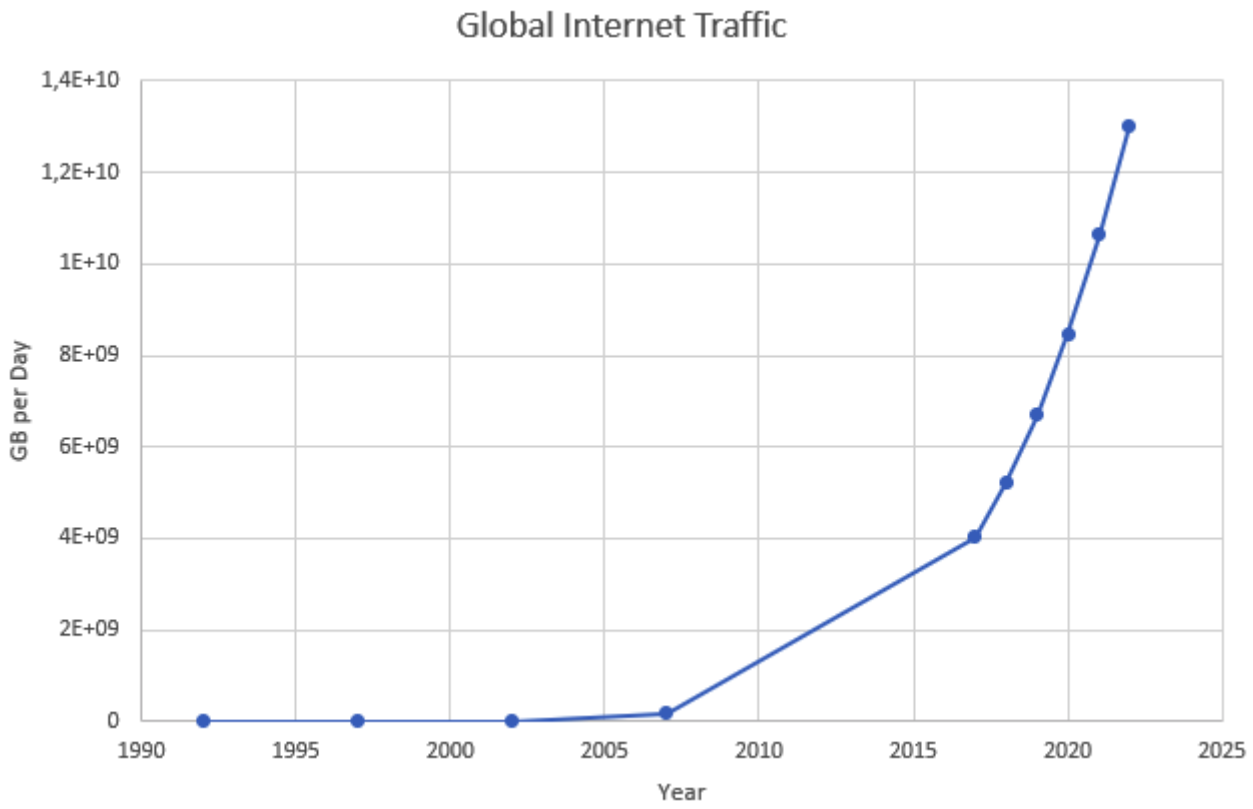


Figure 1- 7. Global Internet Traffic

has meant that companies are being required to modernize their software systems much more rapidly than they ever have. Along with all this, the onset of commodity computing services such as the public cloud has led to a move away from expensive centralized computing systems to more distributed computing ecosystems. As we attempt building our most complex solutions, monoliths are being replaced by an environ of distributed, collaborating microservices. Modern philosophies and practices such as automated testing, architecture fitness functions, continuous integration, continuous delivery, devops, security automation, infrastructure as code, to name a few, are disrupting the way we deliver software solutions.

All these advances introduce their own share of complexity. Instead of attempting to control the amount of complexity, there is a need to embrace and cope with it.

Optimizing the feedback loop

As we enter an age of encountering our most complex business problems, we need to embrace new ways of thinking, a development philosophy and an arsenal of techniques to iteratively evolve mature software solutions that will stand the test of time. We need better ways of communicating, analyzing problems, arriving at a collective understanding, creating and modeling abstractions, and then implementing, enhancing the solution.

To state the obvious — we're all building software with seemingly brilliant business ideas on one side and our ever-demanding customers on the other, as shown here:

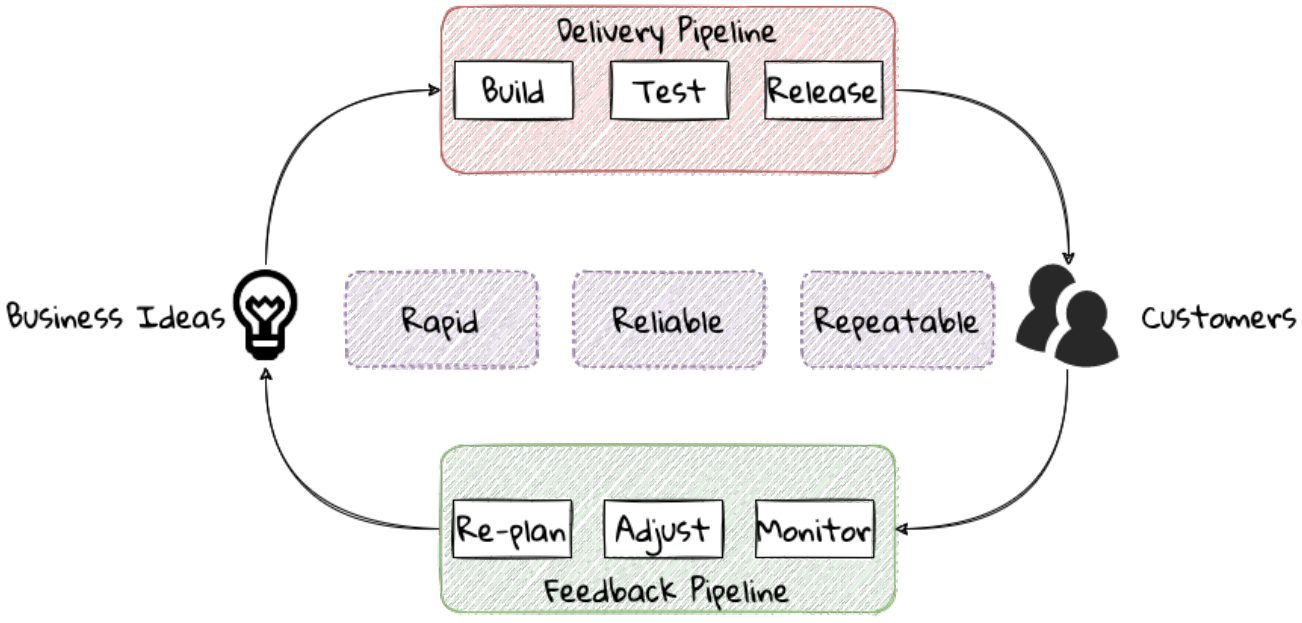


Figure 1- 8. The software delivery continuum

In between, we have two chasms to cross — the *delivery pipeline* and the *feedback pipeline*. The delivery pipeline enables us to put software in the hands of our customers, whereas the feedback pipeline allows us to adjust and adapt. As we can see, this is a continuum. And if we are to build better, more valuable software, this continuum, this potentially infinite loop has to be optimized!

To optimize this loop, we need three characteristics to be present: we need to be fast, we need to be reliable, and we need to do this over and over again. In other words, we need to be rapid, reliable and repeatable — all at the same time!! Take any one of these away, and it just won't sustain.

Domain-driven design promises to provide answers on how to do this in a systematic manner. In the upcoming section, and indeed the rest of this book, we will examine what DDD is and why it is indispensable when working to provide solutions for non-trivial problems in today's world of massively distributed teams and applications.

What is Domain-Driven Design?

> Life is really simple, but we insist on making it complicated.
>
> — Confucius

In the previous section, we saw how a myriad of reasons coupled with system complexity get in the way of software project success. The idea of domain-driven design, originally conceived by Eric Evans in his 2003 book, is an approach to software development that focuses on expressing software solutions in the form of a model that closely embodies the core of the problem being solved. It provides a set of principles and systematic techniques to analyze, architect and implement software solutions in a manner that enhances chances of success.

While Evans' work is indeed seminal, ground-breaking, and way ahead of its time, it is not prescriptive at all. This is a strength in that it has enabled evolution of DDD beyond what Evans had originally conceived at the time. On the other hand, it also makes it extremely hard to define what DDD actually encompasses, making practical application a challenge. In this section, we will look at some foundational terms and concepts behind domain-driven design. Elaboration and practical application of these concepts will happen in upcoming chapters of this book.

When encountered with a complex business problem:

1. **Understand the problem**: To have a deep, shared understanding of the problem, it is necessary for business experts and technology experts to collaborate closely. Here we collectively understand what the problem is and why it is valuable to solve. This is termed as the **domain** for the problem.
2. **Break down the problem** into more manageable parts: To keep complexity at manageable levels, break down complex problems into smaller, independently solvable parts. These parts are termed as **subdomains**. It may be necessary to further break down subdomains where the subdomain is still too complex. Assign explicit boundaries to limit the functionality of each subdomain. This boundary is termed as the **bounded context** for that subdomain. It may also be convenient to think of the subdomain as a concept that makes more sense to the domain experts (in the problem space), whereas the bounded context is a concept that makes more sense to the technology experts (in the solution space).
3. For each of these bounded contexts:
    a. **Agree on a shared language**: Formalize the understanding by establishing a shared language that is applicable unambiguously within the bounds of the subdomain. This shared language is termed as the ubiquitous language of the domain.
    b. **Express understanding in shared models**: In order to produce working software, express the ubiquitous language in the form of shared models. This model is termed as the **domain model**. There may exist multiple variations of this model, each meant to clarify a specific aspect of the solution. For example, a process model, a sequence diagram, working code, a

deployment topology, etc.

4. **Embrace incidental complexity** of the problem: It is important to note that it is not possible to shy away from the essential complexity of a given problem. By breaking down the problem into subdomains and bounded contexts, we are attempting to distribute it (more or less) evenly across more manageable parts.
5. **Continuously evolve** for greater insight: It is important to understand that the above steps are not a one-time activity. Businesses, technologies, processes and our understanding of these evolve, it is important for our shared understanding to remain in sync with these models through continuous refactoring.

A pictorial representation of the essence of domain-driven design is expressed here:

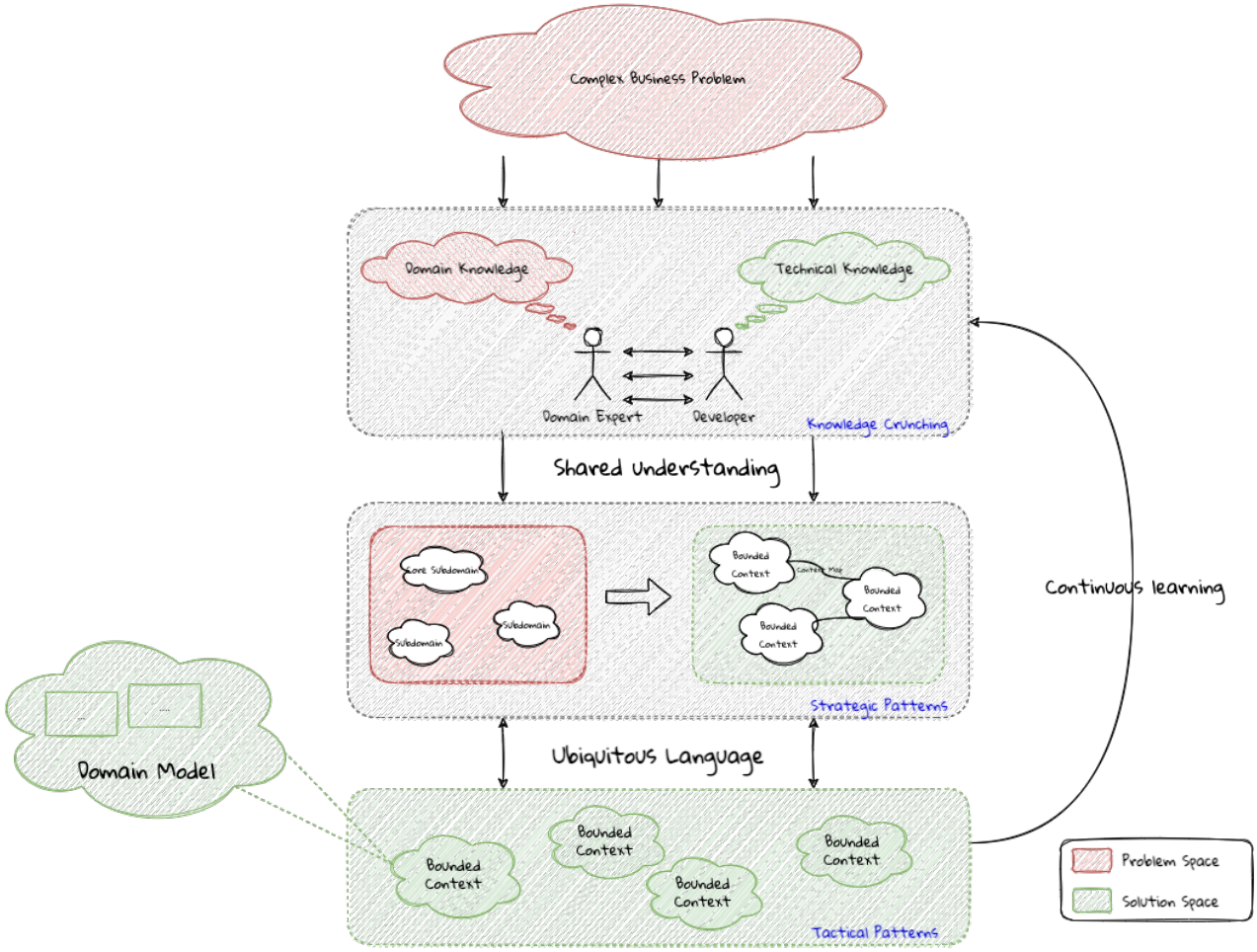


Figure 1- 9. Essence of DDD

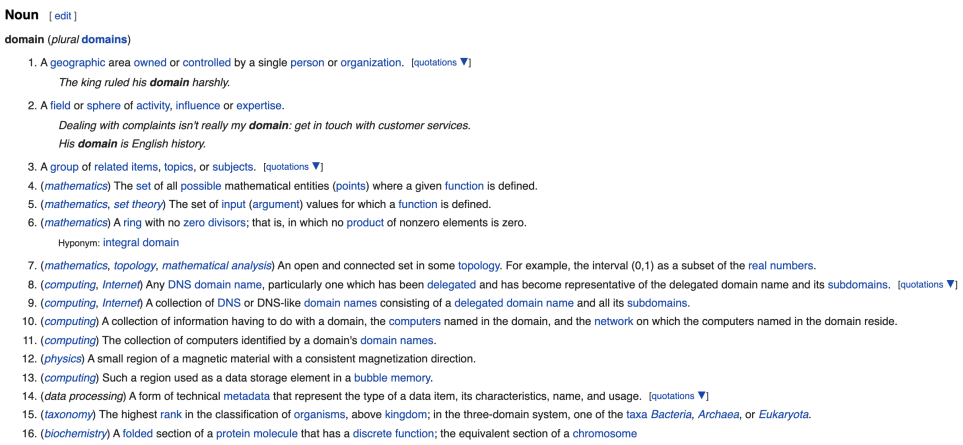
We appreciate that this is quite a whirlwind introduction to the subject of domain-driven design.

Understanding the problem using strategic design

In this section, let's demystify some commonly used concepts and terms when working with domain-driven design. First and foremost, we need to understand what we mean by the first "D" — **domain**.

What is a domain?

The foundational concept when working with domain-driven design is the notion of a domain.But what exactly is a domain?The word domain, which has its origins in the 1600s to the Old French word *domaine* (power), Latin word *dominium* (property, right of ownership) is a rather confusing word.Depending on who, when, where and how it is used, it can mean different things:



Noun [edit]

domain (*plural* **domains**)

1. A geographic area owned or controlled by a single person or organization. [quotations ▼]
   *The king ruled his **domain** harshly.*
2. A field or sphere of activity, influence or expertise.
   *Dealing with complaints isn't really my **domain**: get in touch with customer services.*
   *His **domain** is English history.*
3. A group of related items, topics, or subjects. [quotations ▼]
4. (*mathematics*) The set of all possible mathematical entities (points) where a given function is defined.
5. (*mathematics*, *set theory*) The set of input (argument) values for which a function is defined.
6. (*mathematics*) A ring with no zero divisors; that is, in which no product of nonzero elements is zero.
   Hyponym: integral domain
7. (*mathematics*, *topology*, *mathematical analysis*) An open and connected set in some topology. For example, the interval (0,1) as a subset of the real numbers.
8. (*computing*, *Internet*) Any DNS domain name, particularly one which has been delegated and has become representative of the delegated domain name and its subdomains. [quotations ▼]
9. (*computing*, *Internet*) A collection of DNS or DNS-like domain names consisting of a delegated domain name and all its subdomains.
10. (*computing*) A collection of information having to do with a domain, the computers named in the domain, and the network on which the computers named in the domain reside.
11. (*computing*) The collection of computers identified by a domain's domain names.
12. (*physics*) A small region of a magnetic material with a consistent magnetization direction.
13. (*computing*) Such a region used as a data storage element in a bubble memory.
14. (*data processing*) A form of technical metadata that represent the type of a data item, its characteristics, name, and usage. [quotations ▼]
15. (*taxonomy*) The highest rank in the classification of organisms, above kingdom; in the three-domain system, one of the taxa *Bacteria*, *Archaea*, or *Eukaryota*.
16. (*biochemistry*) A folded section of a protein molecule that has a discrete function; the equivalent section of a chromosome

Figure 1- 10. **Domain***: Means many things depending on context*

In the context of a business however, the word domain covers the overall scope of its primary activity — the service it provides to its customers.This is also referred as the ***problem domain***.For example, Tesla operates in the domain of electric vehicles, Netflix provides online movies and shows, while McDonald's provides fast food.Some companies like Amazon, provide services in more than one domain — online retail, cloud computing, among others.The domain of a business (at least the successful ones) almost always encompasses fairly complex and abstract concepts.To cope with this complexity, it is usual to decompose these domains into more manageable pieces called subdomains.Let us understand subdomains in more detail next.

What is a subdomain?

At its essence, Domain-driven design provides means to tackle complexity.Engineers do this by breaking down complex problems into more manageable ones called **subdomains**.This facilitates better understanding and makes it easier to arrive at a solution.For example, the online retail domain may be divided into subdomains such as product, inventory, rewards, shopping cart, order management, payments, shipping, etc. as shown below:

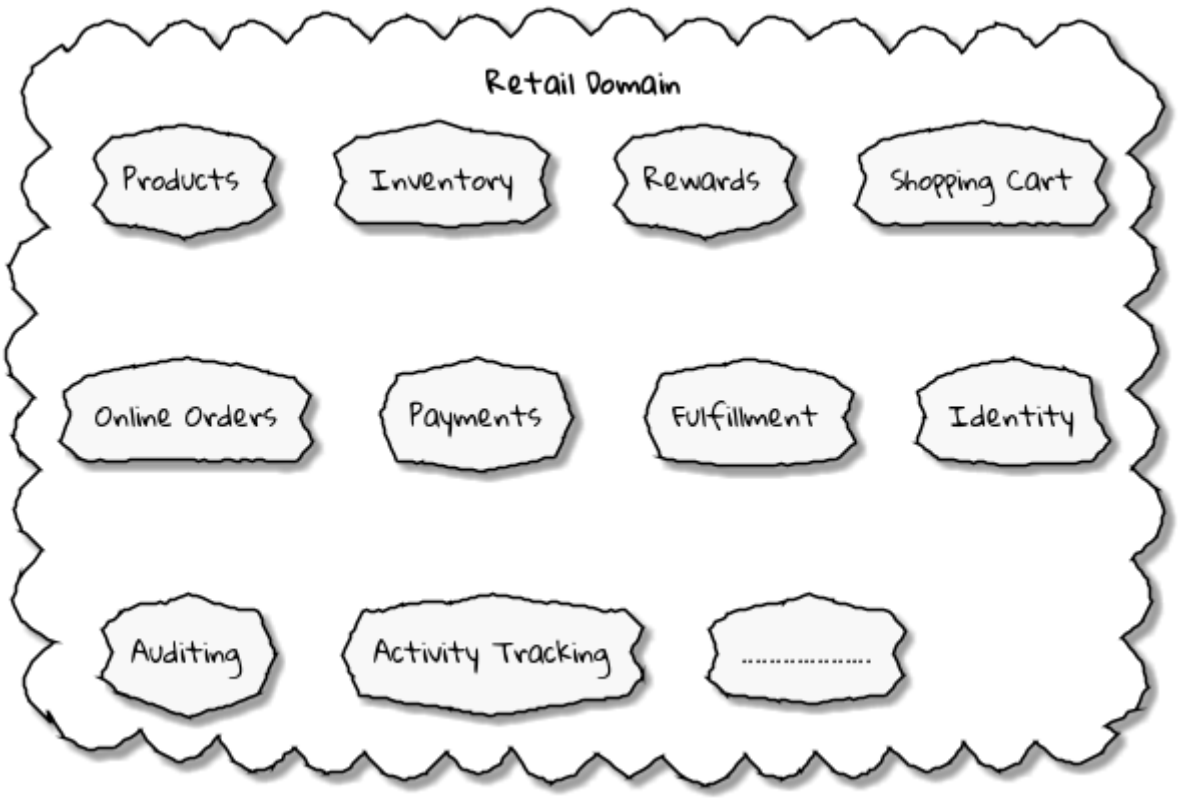


Figure 1- 11. Subdomains in the Retail domain

In certain businesses, subdomains themselves may turn out to become very complex on their own and may require further decomposition. For instance, in the retail example above, it may be required to break the products subdomain into further constituent subdomains such as catalog, search, recommendations, reviews, etc. as shown below:

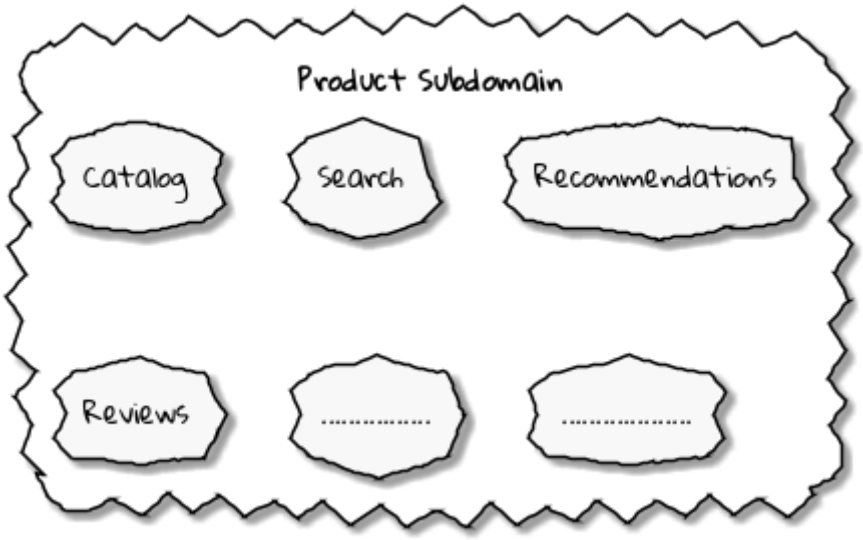


Figure 1- 12. Subdomains in the Products subdomain

Further breakdown of subdomains may be needed until we reach a level of manageable complexity.Domain decomposition is an important aspect of DDD.Let's look at the types of subdomains to understand this better.

The terms domain and subdomains tend to get used interchangeably quite often.This can be confusing to the casual onlooker.Given that sub(domains) tend to be quite complex and hierarchical, a subdomain can be a domain in its own right.

Types of subdomains

Breaking down a complex domain into more manageable subdomains is a great thing to do.However, not all subdomains are created equal.With any business, the following three types of subdomains are going to be encountered:

- **Core**: The main focus area for the business.This is what provides the biggest differentiation and value.It is therefore natural to want to place the most focus on the core subdomain.In the retail example above, shopping cart and orders might be the biggest differentiation — and hence may form the core subdomains for that business venture.It is prudent to implement core subdomains in-house given that it is something that businesses will desire to have the most control over.In the online retail example above, the business may want to focus on providing an enriched experience to place online orders.This will make the *online orders* and *shopping cart* part of the core subdomain.
- **Supporting**: Like with every great movie, where it is not possible to create a masterpiece without a solid supporting cast, so it is with supporting or auxiliary subdomains.Supporting subdomains are usually very important and very much required, but may not be the primary focus to run the business.These supporting subdomains, while necessary to run the business, do not usually offer a significant competitive advantage.Hence, it might be even fine to completely outsource this work or use an off-the-shelf solution as is or with minor tweaks.For the retail example above, assuming that online ordering is the primary focus of this business, catalog management may be a supporting subdomain.
- **Generic**: When working with business applications, one is required to provide a set of capabilities **not** directly related to the problem being solved.Consequently, it might suffice to just make use of an off-the-shelf solution.For the retail example above, the identity, auditing and activity tracking subdomains might fall in that category.

It is important to note that the notion of core vs. supporting vs. generic subdomains is very context specific.What is core for one business may be supporting or generic for another.Identifying and distilling the core domain requires deep understanding and experience of what problem is being attempted to be solved.

Given that the core subdomain establishes most of the business differentiation, it will be prudent to devote the most amount of energy towards maintaining that differentiation. This is illustrated in the core domain chart here:

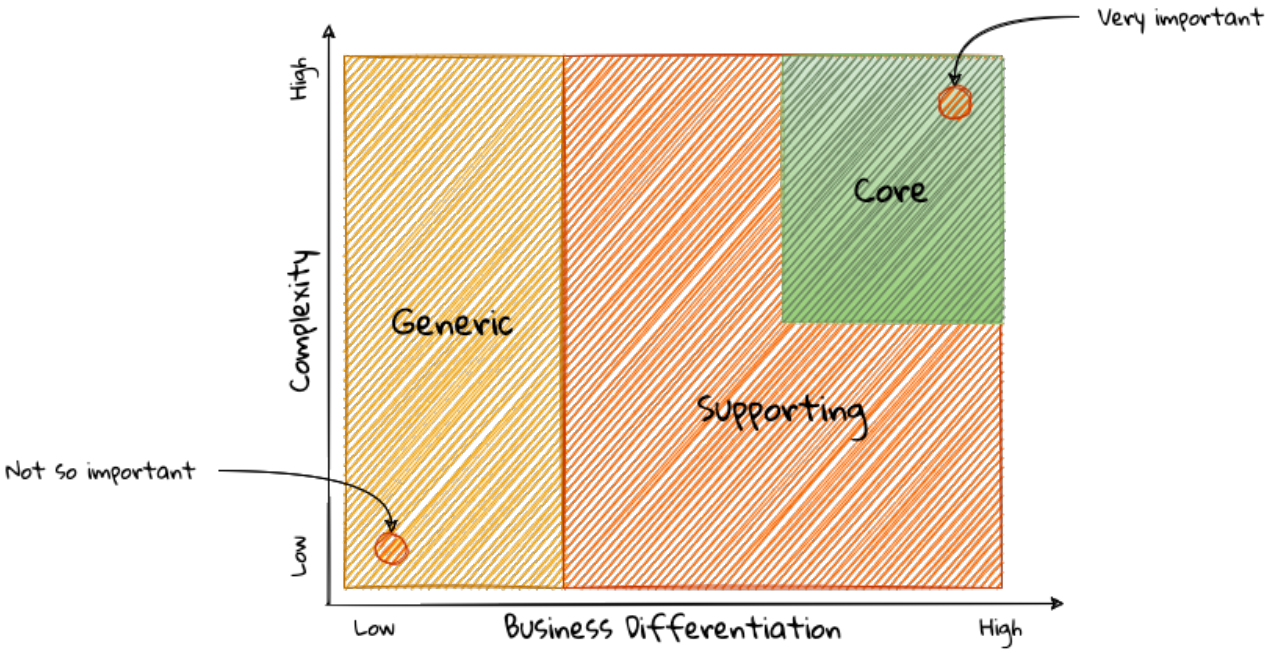


Figure 1- 13. Importance of subdomains

Over a period of time, it is only natural that competitors will attempt to emulate your successes. Newer, more efficient methods will arise, reducing the complexity involved, disrupting your core. This may cause the notion of what is currently core, to shift and become a supporting or generic capability as depicted here:

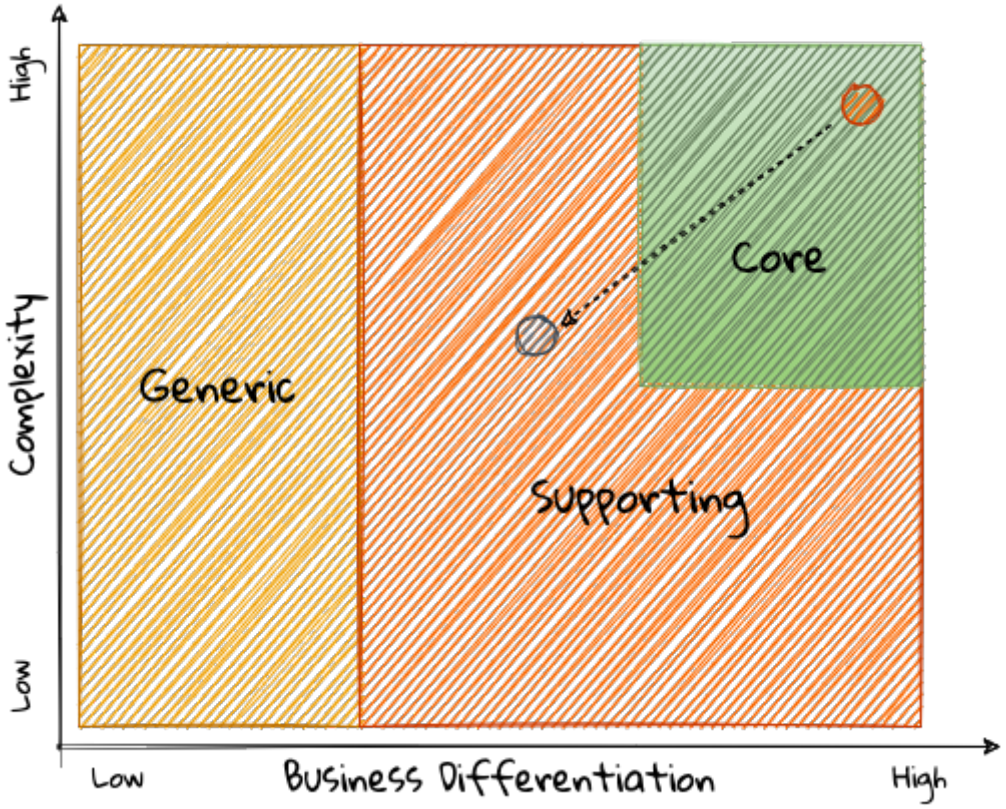


Figure 1- 14. Core domain erosion

To continue running a successful operation, it is required to constantly innovate in the core. For example, when AWS started the cloud computing business, it only provided simple infrastructure

(IaaS) solutions. However, as competitors like Microsoft, Google and others started to catch up, AWS has had to provide several additional value-added services (for example, PaaS, SaaS, etc).

As is evident, this is not just an engineering problem. It requires deep understanding of the underlying business. That's where domain experts can play a significant role.

Domain and technical experts

Any modern software team requires expertise in at least two areas — the functionality of the domain and the art of translating it into high quality software. At most organizations, these exist as at least two distinct groups of people.

Domain experts — those who have a deep and intimate understanding of the domain. Domain experts are subject-matter experts (SMEs) who have a very strong grasp of the business. Domain experts may have varying degrees of expertise. Some SMEs may choose to specialize in specific subdomains, while others may have a broader understanding of how the overall business works.

Technical experts on the other hand, enjoy solving specific, quantifiable computer science problems. Often, technical experts do not feel it worth their while understanding the context of the business they work in. Rather, they seem overly eager to only enhance their technical skills that are a continuation of their learnings in academia.

While the domain experts specify the **why** and the ***what***, technical experts, (software engineers) largely help realize the ***how***. Strong collaboration and synergy between both groups is absolutely essential to ensure sustained high performance and success.

A divide originating in language

While strong collaboration between these groups is necessary, it is important to appreciate that these groups of people seem to have distinct motivations and differences in thinking. Seemingly, this may appear to be restricted to simple things like differences in their day-to-day language. However, deeper analysis usually reveals a much larger divide in aspects such as goals, motivations etc. This is illustrated in the picture here:

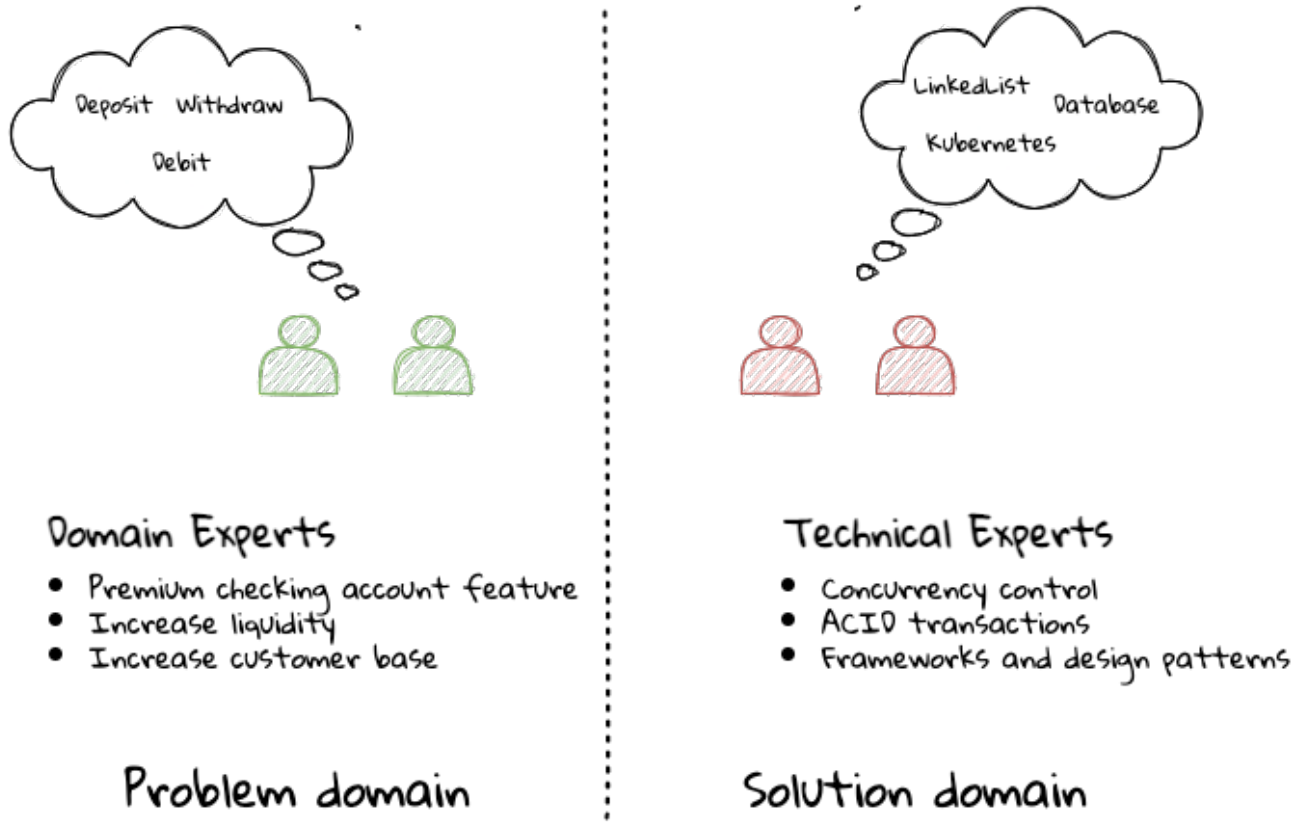


Figure 1- 15. Divide originating in language

But this is a book primarily focused towards technical experts. Our point is that it is not possible to be successful by just working on technically challenging problems without gaining a sound understanding of the underlying business context.

Every decision we take regarding the organization, be it requirements, architecture, code, etc. has business and user consequences. In order to conceive, architect, design, build and evolve software effectively, our decisions need to aid in creating the optimal business impact. As mentioned above, this can only be achieved if we have a clear understanding of the problem we intend to solve. This leads us to the realization that there exist two distinct *domains* when arriving at the solution for a problem:

The use of the word domain in this context is made in an abstract sense — not to be confused with the concept of the business domain introduced earlier.

Problem domain

A term that is used to capture information that simply defines the problem while consciously avoiding any details of the solution. It includes details like **why** we are trying to solve the problem, **what** we are trying to achieve and **how** it needs to be solved. It is important to note that the *why*, *what* and *how* are from the perspective of the customers/stakeholders, not from the perspective of the engineers providing software solutions to the problem.

Consider the example of a retail bank which already provides a checking account capability for their customers. They want access to more liquid funds. To achieve that, they need to encourage customers to maintain higher account balances. To do that, they are looking to introduce a new product called the *premium checking account* with additional features like higher interest rates,

overdraft protection, no-charge ATM access, etc. The problem domain expressed in the form of why, what and how is shown here:

Table 2. Problem domain: why, what and how

Question	Answer
Why	Bank needs access to more liquid funds
What	Have customers maintain higher account balances
How	By introducing a new product — the premium checking account with enhanced features

Now that we have defined the problem and the motivations surrounding it, let's examine how it can inform the solution.

Solution domain

A term used to describe the environment in which the solution is developed. In other words, the process of translating requirements into working software (this includes design, development, testing, deployment, etc). Here the emphasis is on the *how* of the problem being solved from a software implementation perspective. However, it is very difficult to arrive at a solution without having an appreciation of the why and the what.

Building on the previous premium checking account example, the code-level solution for this problem may look something like this:

```
class PremiumCheckingAccountFactory {

    Account openPremiumCheckingAccount(Applicant applicant,
                                       MonetaryAmount initialAmount) {

        Salary salary = checkEmployed(applicant);

        if (salary.isBelowThreshold()) {
            throw new InsufficientIncomeException(applicant);
        }

        Account account = Account.createFor(applicant);
        account.deposit(initialAmount);
        account.activate();
        return account;
    }
}
```

This likely appears like a significant leap from a problem domain description, and indeed it is. Before a solution like this can be arrived at, there may need to exist multiple levels of refinement of the problem. As mentioned in the previous chapter, this process of refinement is usually messy and may lead to inaccuracies in the understanding of the problem, resulting in a solution that may be good (for example, one that is sound from an engineering, software architecture standpoint), but

not one that solves the problem at hand. Let's look at how we can continuously refine our understanding by closing the gap between the problem and the solution domain.

Promoting a shared understanding using a ubiquitous language

Previously, we saw how organizational silos can result in valuable information getting diluted. At a credit card company I used to work with, the words plastic, payment instrument, account, PAN (Primary Account Number), BIN (Bank Identification Number), card were all used by different team members to mean the exact same thing - the ***credit card*** when working in the same area of the application. On the other hand, a term like ***user*** would be used to sometimes mean a customer, a relationship manager, a technical customer support employee. To make matters worse, a lot of these muddled use of terms got implemented in code as well. While this might feel like a trivial thing, it had far-reaching consequences. Product experts, architects, developers, all came and went, each regressively contributing to more confusion, muddled designs, implementation and technical debt with every new enhancement — accelerating the journey towards the dreaded, unmaintainable, big ball of mud.

DDD advocates breaking down these artificial barriers, and putting the domain experts and the developers on the same level footing by working collaboratively towards creating what DDD calls a ***ubiquitous language*** — a shared vocabulary of terms, words, phrases to continuously enhance the collective understanding of the entire team. This phraseology is then used actively in every aspect of the solution: the everyday vocabulary, the designs, the code — in short by **everyone** and **everywhere**. Consistent use of the common ubiquitous language helps reinforce a shared understanding and produce solutions that better reflect the mental model of the domain experts.

Evolving a domain model and a solution

The ubiquitous language helps establish a consistent albeit informal lingo among team members. To enhance understanding, this can be further refined into a formal set of abstractions — a ***domain model*** to represent the solution in software. When a problem is presented to us, we subconsciously attempt to form mental representations of potential solutions. Further, the type and nature of these representations (models) may differ wildly based on factors like our understanding of the problem, our backgrounds and experiences, etc. This implies that it is natural for these models to be different. For example, the same problem can be thought of differently by various team members as shown here:

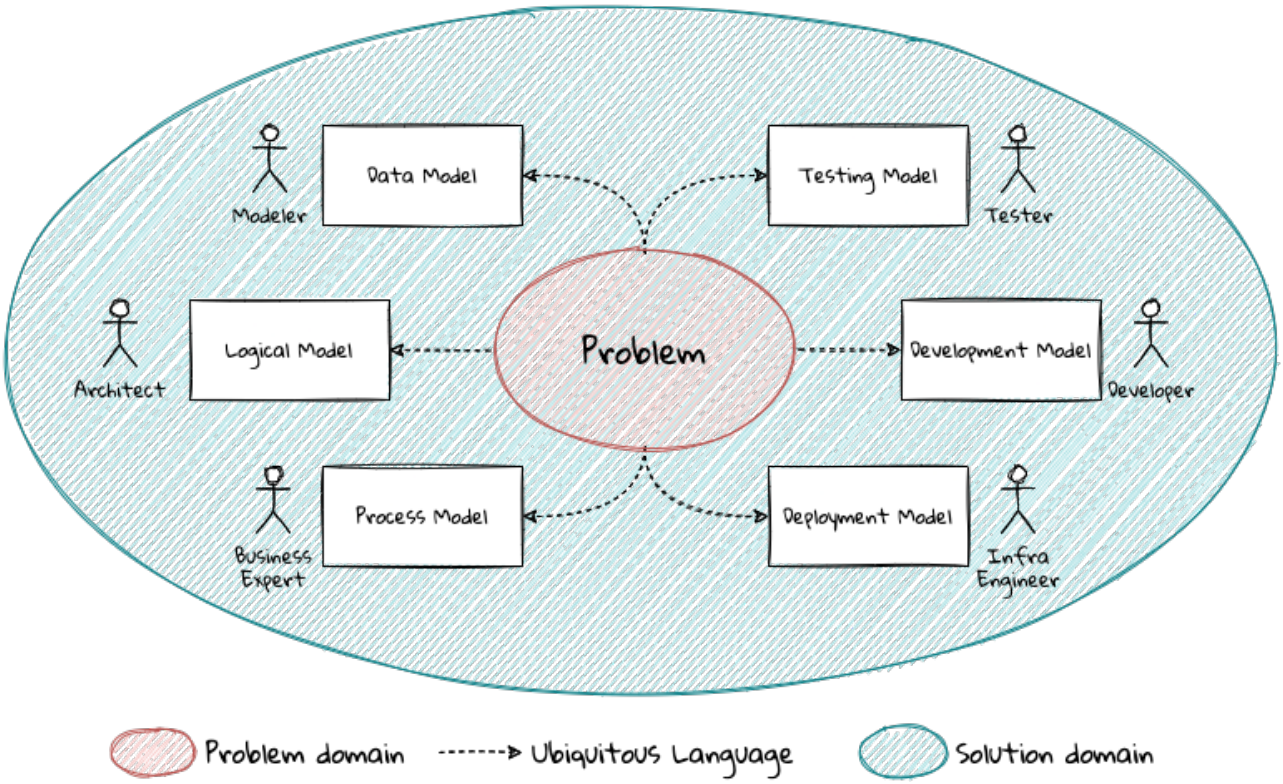


Figure 1- 16. Multiple models to represent the solution to the problem using the ubiquitous language

As illustrated here, the business expert may think of a process model, whereas the test engineer may think of exceptions and boundary conditions to arrive at a test strategy and so on.

The illustration above is to depict the existence of multiple models. There may be several other perspectives, for example, a customer experience model, an information security model, etc. which are not depicted.

Care should be taken to retain focus on solving the business problem at hand at all times. Teams will be better served if they expend the same amount of effort modeling business logic as the technical aspects of the solution. To keep accidental complexity in check, it will be best to isolate the infrastructure aspects of the solution from this model. These models can take several forms, including conversations, whiteboard sessions, documentation, diagrams, tests and other forms of architecture fitness functions. It is also important to note that this is **not** a one-time activity. As the business evolves, the domain model and the solution will need to keep up. This can only be achieved through close collaboration between the domain experts and the developers at all times.

Scope of domain models and the bounded context

When creating domain models, one of the dilemmas is in deciding how to restrict the scope of these models. One can attempt to create a single domain model that acts as a solution for the entire problem. On the other hand, we may go the route of creating extremely fine-grained models that cannot exist meaningfully without having a strong dependency on others. There are pros and cons in going each way. Whatever be the case, each solution has a scope — bounds to which it is confined to. This boundary is termed as a **bounded context**.

There seems to exist a lot of confusion between the terms subdomains and bounded contexts. What is the difference? It turns out that subdomains are problem space concepts whereas bounded

contexts are solution space concepts. This is best explained through the use of an example. Let's consider the example of a fictitious Acme bank that provides two products: credit cards and retail bank. This may decompose to the following subdomains as depicted here:

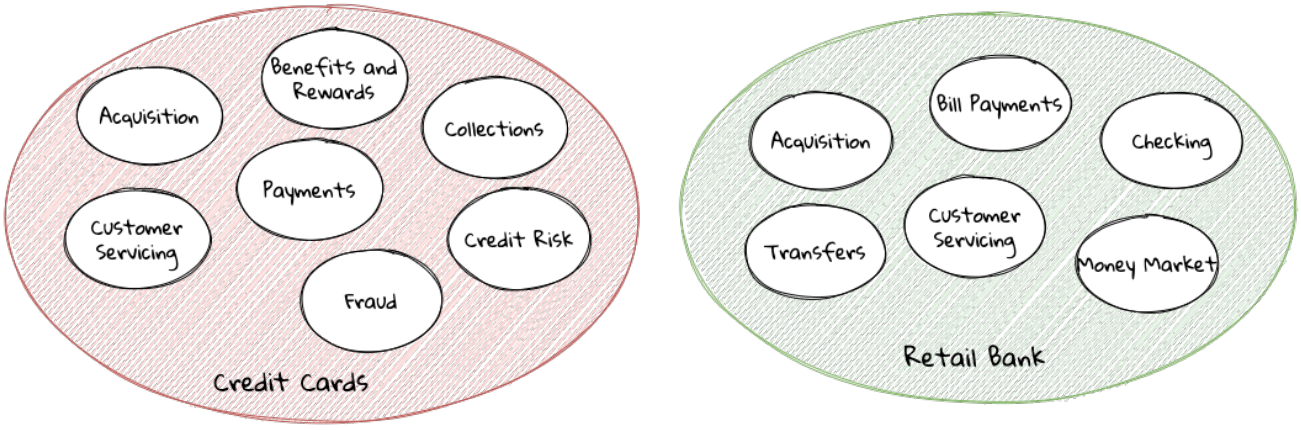


Figure 1- 17. Banking subdomains at Acme bank

When creating a solution for the problem, many possible solution options exist. We have depicted a few options here:

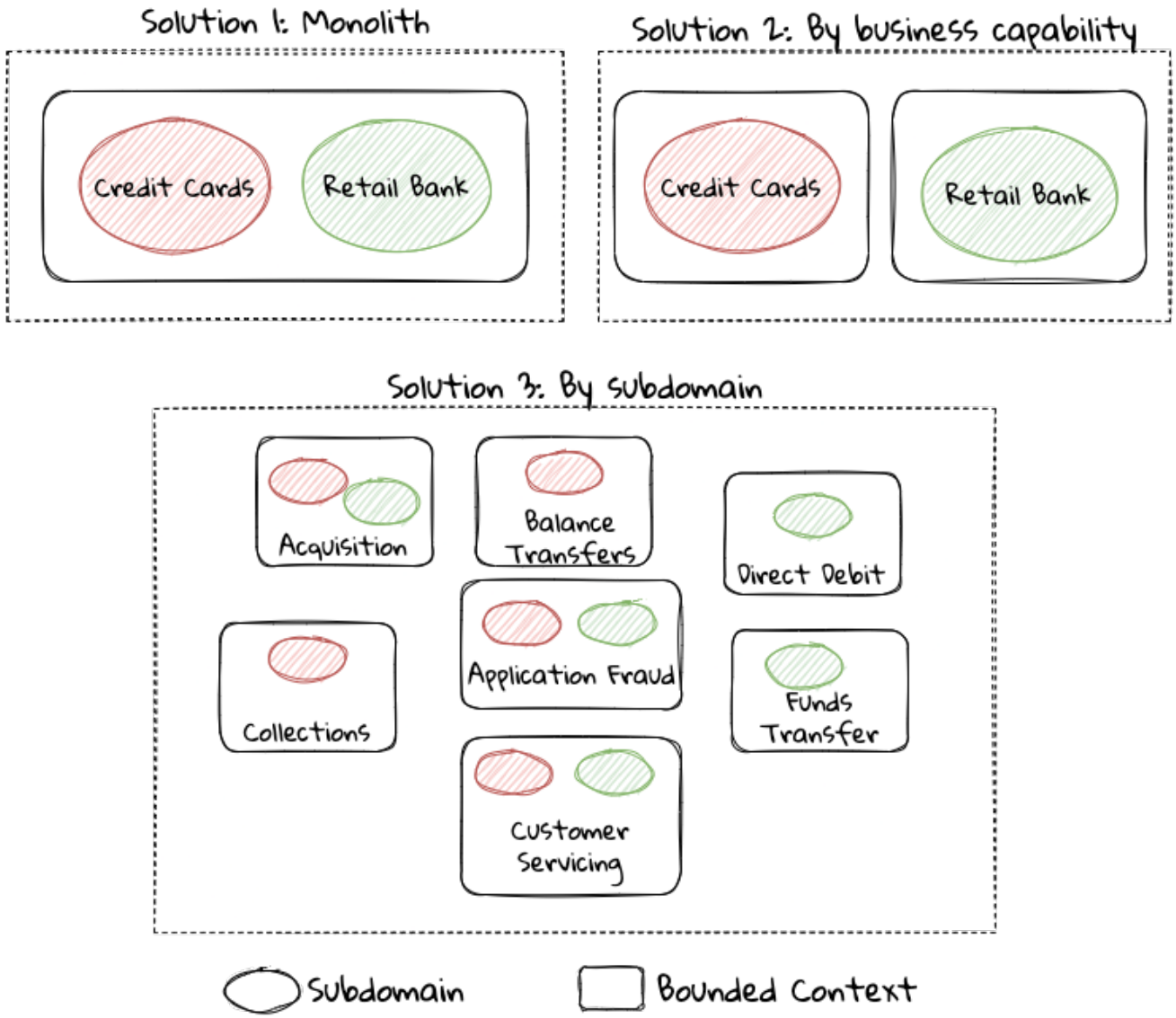


Figure 1- 18. Bounded contexts options at Acme bank

These are just a few examples of decomposition patterns to create bounded contexts. The exact set of patterns one may choose to use may vary depending on currently prevailing realities like:

- Current organizational structures
- Domain experts' responsibilities
- Key activities and pivotal events
- Existing applications

Conway's Law asserts that organizations are constrained to produce application designs which are copies of their communication structures. Your current organizational structures may not be optimally aligned to your desired solution approach. The **inverse Conway maneuver**[1] may be applied to achieve isomorphism with the business architecture.

Whatever be the method used to decompose a problem into a set of bounded contexts, care should be taken to make sure that the coupling between them is kept as low as possible.

While bounded contexts ideally need to be as independent as possible, they may still need to communicate with each other. When using domain-driven design, the system as a whole can be represented as a set of bounded contexts which have relationships with each other. These relationships define how these bounded contexts can integrate with each other and are called ***context maps***. A sample context map is shown here.

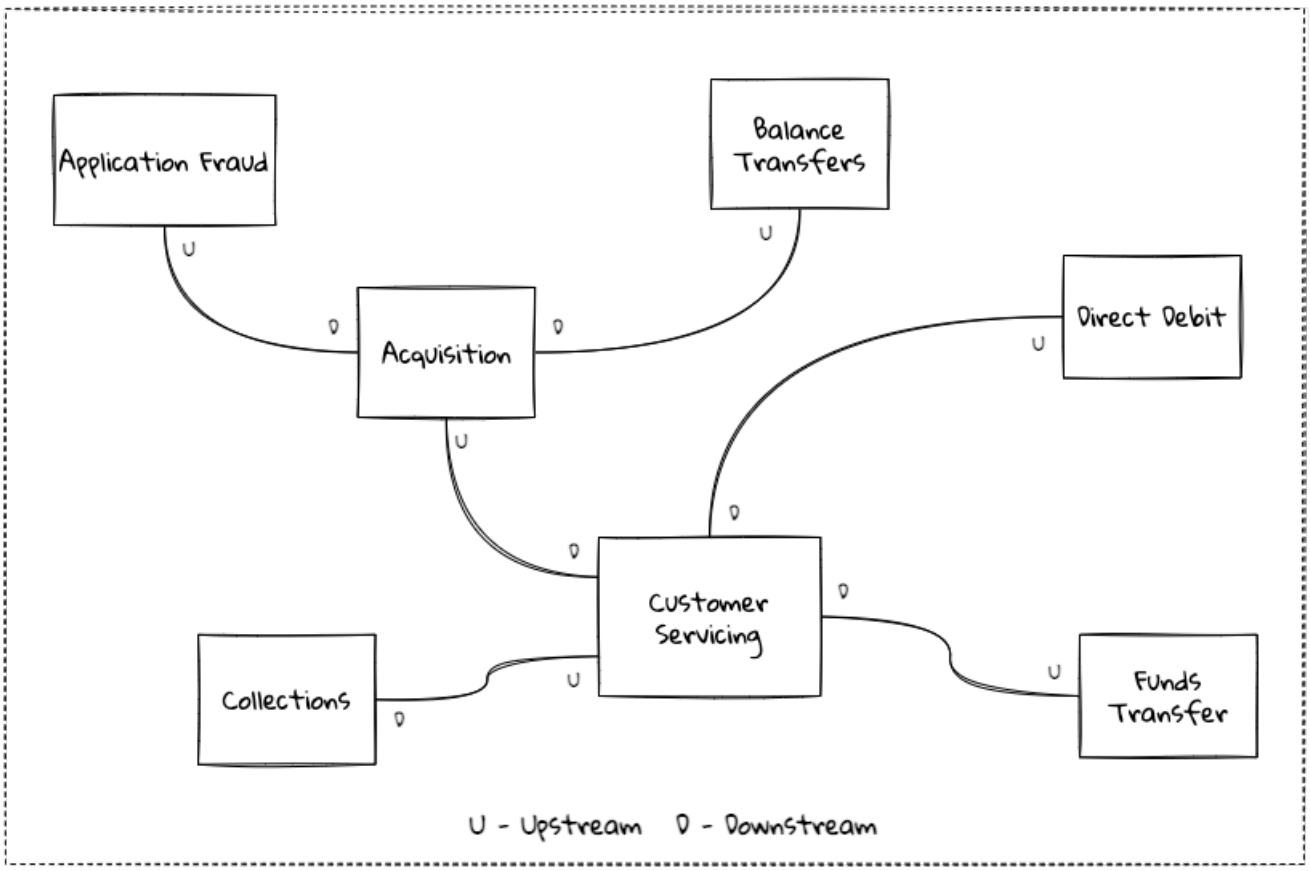


Figure 1- 19. Sample context map for Acme bank

The context map shows the bounded contexts the relationship between them. These relationships

can be a lot more nuanced than what is depicted here. We will discuss more details on context maps and communication patterns in Chapter 9: Integrating with external systems.

We have now covered a catalog of concepts that are core to the strategic design tenets of domain-driven design. Let's look at some tools that can help expedite this process.

In subsequent chapters we will reinforce all the concepts introduced here in a lot more detail.

In the next section, we will look at why the ideas of DDD, introduced all those years ago, are still very relevant. If anything, we will look at why they are becoming even more relevant now than ever.

Implementing the solution using tactical design

In the previous section, we have seen how we can arrive at a shared understanding of the problem using the strategic design tools. We need to use this understanding to create a solution. DDD's tactical design aspects, tools and techniques help translate this understanding into working software. Let's look at these aspects in detail. In part 2 of the book, we will apply these to solve a real-world problem.

It is convenient to think of the tactical design aspects as depicted in this picture:

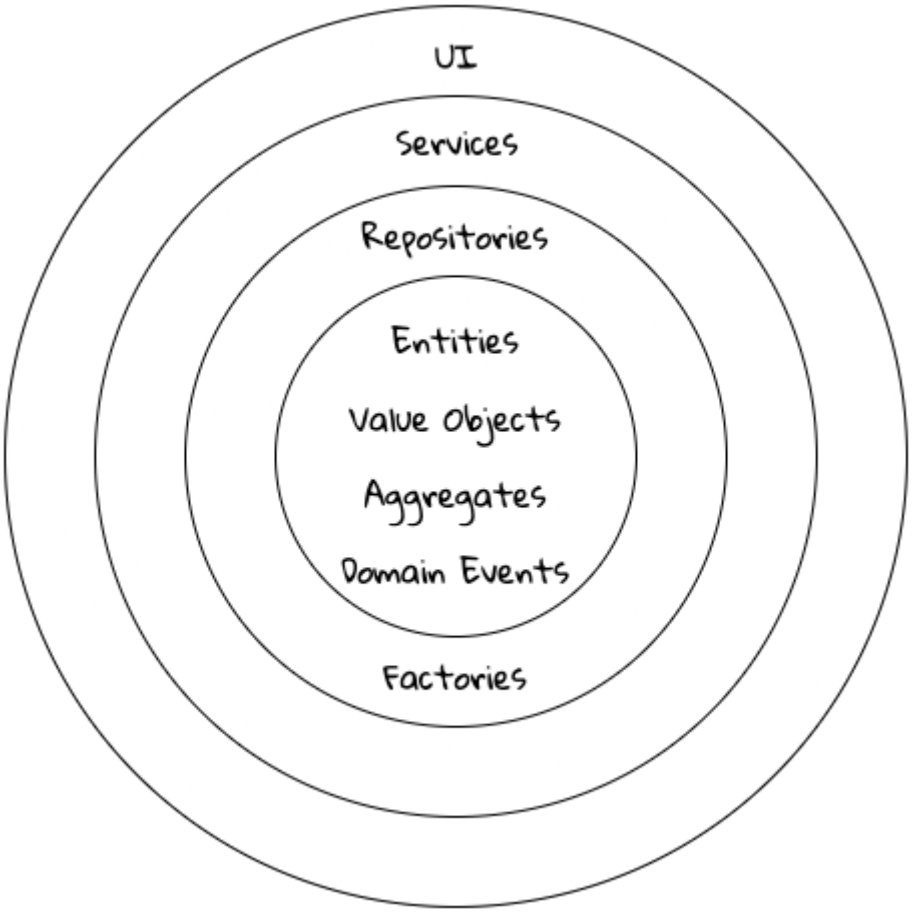


Figure 1- 20. The elements of DDD's tactical design

Let's look at the definitions of these elements.

Value objects

Value objects are immutable objects that encapsulate the data and behavior of one or more related attributes. It may be convenient to think of value objects as named primitives. For example, consider a `MonetaryAmount` value object. A simple implementation can contain two attributes — an *amount* and a *currency code*. This allows encapsulation of behavior such as adding two `MonetaryAmount` objects safely as shown here:

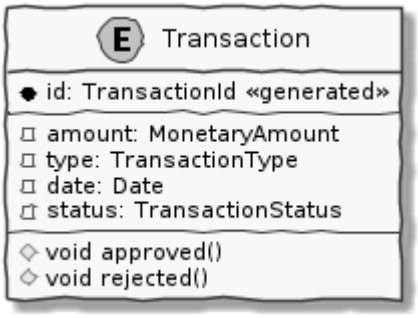


Figure 1- 21. A simple `MonetaryAmount` value object

The effective use of value objects helps protect from the primitive obsession[2] antipattern, while increasing clarity. It also allows composing higher level abstractions using one or more value objects. It is important to note that value objects do not have the notion of identity. That is, two value having the same value are treated equal. So two `MonetaryAmount` objects having the same `amount` and `currency code` will be considered equal. Also, it is important to make value objects immutable. That is, a need to change any of the attributes should result in the creation of a new attribute.

It is easy to dismiss the use of value objects as a mere engineering technique, but the consequences of (not) using them can be far-reaching. In the `MonetaryAmount` example above, it is possible for the `amount` and `currency code` to exist as independent attributes. However, the use of the `MonetaryAmount` enforces the notion of the *ubiquitous language*. Hence, we recommend the use of value objects as a default instead of using primitives.

Critics may be quick to point out problems such as class explosion and performance issues. But in our experience, the benefits usually outweigh the costs. But it may be necessary to re-examine this approach if problems occur.

Entities

An entity is an object with a **unique identity** and **encapsulates** the data and behaviour of its attributes. It may be convenient to view entities as a collection of other entities and value objects that need to be grouped together. A very simple example of an entity is shown here:

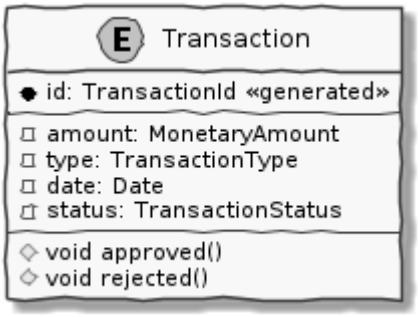


Figure 1- 22. A simple depiction of `Transaction` entity

In contrast to a value object, entities have the notion of a unique identifier. This means that two `Transaction` entities having the same underlying values, but having a different identifier (`id`) value, will be considered different. On the other hand, two entity instances having the same value for the identifier are considered equal. Furthermore, unlike value objects, entities are mutable. That is, their attributes can and will change over time.

The concept of value objects and entities depends on the context within which they are used. In an order management system, the `Address` may be implemented as a value object in the *E-Commerce* bounded context, whereas it may be needed to be implemented as an entity in the *Order Fulfillment* bounded context.

It is common to collectively refer to entities and value objects as *domain objects*.

Aggregates

As seen above, entities are hierarchical, in that they can be composed of one more children. Fundamentally, an aggregate:

- Is an entity usually composed of other child entities and value objects.
- Encapsulates access to child entities by exposing behavior (usually referred to as *commands*).
- Is a boundary that is used to enforce business invariants (rules) in a consistent manner.
- Is an entry point to get things done within a bounded context.

Consider the example of a `CheckingAccount` aggregate:

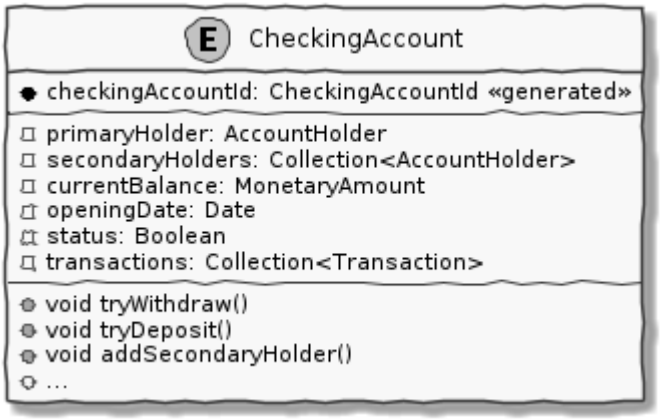


Figure 1- 23. A simple depiction of a `CheckingAccount` aggregate

Note how the `CheckingAccount` is composed using the `AccountHolder` and Transaction` entities among other things. In this example, let's assume that the overdraft feature (ability to hold a negative account balance) is only available for high net-worth individuals (HNI). Any attempt to change the `currentBalance` needs to occur in the form of a unique `Transaction` for audit purposes — irrespective of its outcome. For this reason, the `CheckingAccount` aggregate makes use of the `Transaction` entity. Although the `Transaction` has `approve` and `reject` methods as part of its interface, only the aggregate has access to these methods. In this way, the aggregate enforces the business invariant while maintaining high levels of encapsulation. A potential implementation of the `tryWithdraw` method is shown here:

```
class CheckingAccount {
    private AccountHolder primaryHolder;                          ①
private Collection<Transaction> transactions;                     ①
    private MonetaryAmount currentBalance;                        ①
    // Other code omitted for brevity

    void tryWithdraw(MonetaryAmount amount) {                     ②
        MonetaryAmount newBalance = this.currentBalance.subtract(amount);
        Transaction transaction = add(Transaction.withdrawal(this.id, amount));
        if (primaryHolder.isNotHNI() && newBalance.isOverdrawn()) { ③
        transaction.rejected();
        } else {
            transaction.approved();
            currentBalance = newBalance;
        }
    }
}
```

① The `CheckingAccount` aggregate is composed of child entities and value objects.

② The `tryWithdraw` method acts as a consistency boundary for the operation. Irrespective of the outcome (approved or rejected), the system will remain in a consistent state. In other words, the `currentBalance` can change only within the confines of the `CheckingAccount` aggregate.

③ The aggregate enforces the appropriate business invariant (rule) to allow overdrafts only for HNIs.

Aggregates are also referred to as **aggregate roots**. That is, the object that is at the root of the entity hierarchy. We use these terms synonymously in this book.

Domain events

As mentioned above, aggregates dictate how and when state changes occur. Other parts of the system may be interested in knowing about the occurrence of changes that are significant to the business. For example, an order being placed or a payment being received, etc. *Domain events* are the means to convey that something business significant has occurred. It is important to differentiate between system events and domain events. For example, in the context of a retail bank, a *row was saved* in the database, or a *server ran out of disk space*, etc. may classify as system events, whereas a *deposit was made* to a checking account, *fraudulent activity was detected* on a transaction, etc. could be classified as domain events. In other words, domain events are things that

domain experts care about.

It may be prudent to make use of domain events to reduce the amount of coupling between bounded contexts, making it a critical building block of domain-driven design.

Repositories

Most businesses require durability of data. For this reason, aggregate state needs to be persisted and retrieved when needed. Repositories are objects that enable persisting and loading *aggregate* instances. This is well documented in Martin Fowler's *Patterns of Enterprise Application Architecture* book as part of the repository[3] pattern. It is pertinent to note that we are referring to aggregate repositories here, not just any entity repository. The singular purpose of this repository is to load a **single instance** of an aggregate using its identifier. It is important to note that this repository does not support finding aggregate instances using any other means. This is because, business operations happen as part of manipulating a single instance of the aggregate within its bounded context.

Factories

In order to work with aggregates and value objects, instances of these need to be constructed. In simple cases, it might suffice to use a constructor to do so. However, aggregate and value object instances can become quite complex depending on amount the state they encapsulate. In such cases, it may be prudent to consider delegating object construction responsibilities to a *factory* external to the aggregate/value object. We make use of the static factory method, builder, and dependency injection quite commonly in our day-to-day. Joshua Bloch discusses several variations of this pattern in *Chapter 2: Creating and destroying objects* in his *Effective Java* book.

Services

When working within the confines of a single bounded context, the public interface (commands) of the aggregate provides a natural API. However, more complex business operations may require interacting with multiple bounded contexts and aggregates. In other words, we may find ourselves in situations where certain business operations do not fit naturally with any single aggregate. Even if interactions are limited to a single bounded context, there may be a need to expose that functionality in an implementation-neutral manner. In such cases, one may consider the use of objects termed as *services*. Services come in at least 3 flavors:

1. **Domain services**: To enable coordinating operations among more than one aggregate. For example, transferring money between two checking accounts at a retail bank.
2. **Infrastructure services**: To enable interactions with a utility that is not core to the business. For example, logging, sending emails, etc. at the retail bank.
3. **Application services**: Enable coordination between domain services, infrastructure services and other application services. For example, sending email notifications after a successful inter-account money transfer.

Services can also be stateful or stateless. It is best to allow aggregates to manage state making use of repositories, while allowing services to coordinate and/or orchestrate business flows. In complex cases, there may be a need to manage the state of the flow itself. We will look at more concrete examples in part 2 of this book.

It may become tempting to implement business logic almost exclusively using services — inadvertently leading to the anemic domain model[4] anti-pattern. It is worthwhile striving to encapsulate business logic within the confines of aggregates as a default.

Why is DDD Relevant? Why Now?

> He who has a why to live for can bear almost anyhow.
>
> — Friedrich Nietzsche

In a lot of ways, domain-driven design was way ahead of its time when Eric Evans introduced the concepts and principles back in 2003. DDD seems to have gone from strength to strength. In this section, we will examine why DDD is even more relevant today, than it was when Eric Evans wrote his book on the subject way back in 2003.

Rise of Open Source

Eric Evans, during his keynote address at the Explore DDD conference in 2017, lamented about how difficult it was to implement even the simplest concepts like immutability in value objects when his book had released. In contrast though, nowadays, it's simply a matter of importing a mature, well documented, tested library like Project Lombok or Immutables to be productive, literally in a matter of minutes. To say that open source software has revolutionized the software industry would be an understatement! At the time of this writing, the public maven repository (https://mvnrepository.com) indexes no less than a staggering **18.3 million artifacts** in a large assortment of popular categories ranging from databases, language runtimes to test frameworks and many many more as shown in the chart below:

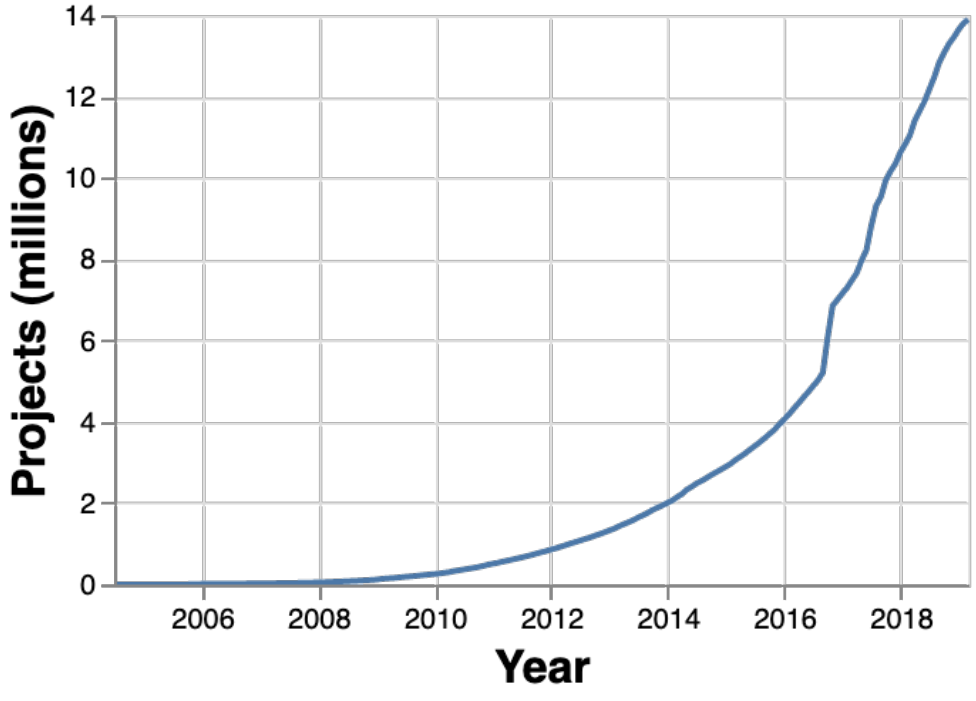


Figure 1- 24. Open source Java over the years. Source: https://mvnrepository.com/

Java stalwarts like the spring framework and more recent innovations like spring boot, quarkus, etc. make it a no-brainer to create production grade applications, literally in a matter of minutes. Furtheremore, frameworks like Axon, Lagom, etc. make it relatively simple to implement advanced architecture patterns such are CQRS, event sourcing, that are very complementary to implementing DDD-based solutions.

Advances in Technology

DDD by no means is just about technology, it could not be completely agnostic to the choices available at the time. 2003 was the heyday of heavyweight and ceremony-heavy frameworks like J2EE (Java 2 Enterprise Edition), EJBs (Enterprise JavaBeans), SQL databases, ORMs (Object Relational Mappers) and the like — with not much choice beyond that when it came to enterprise tools and patterns to build complex software — at least out in the public domain. The software world has evolved and come a very long way from there. In fact, modern game changers like Ruby on Rails and the public cloud were just getting released. In contrast though, we now have no shortage of application frameworks, NoSQL databases, programmatic APIs to create infrastructure components with a lot more releasing with monotonous regularity.

All these innovations allow for rapid experimentation, continuous learning and iteration at pace. These game changing advances in technology have also coincided with the exponential rise of the internet and ecommerce as viable means to carry out successful businesses. In fact the impact of the internet is so pervasive that it is almost inconceivable to launch businesses without a digital component being an integral component. Finally, the consumerization and wide scale penetration of smartphones, IoT devices and social media has meant that data is being produced at rates inconceivable as recent as a decade ago. This means that we are building for and solving the most complicated problems by several orders of magnitude.

Rise of Distributed Computing

There was a time when building large monoliths was very much the default. But an exponential rise in computing technology, public cloud, (IaaS, PaaS, SaaS, FaaS), big data storage and processing volumes, which has coincided with an arguable slowdown in the ability to continue creating faster CPUs, have all meant a turn towards more decentralized methods of solving problems.

[Hilbert InfoGrowth] |

https://upload.wikimedia.org/wikipedia/commons/7/7c/Hilbert_InfoGrowth.png

Figure 1- 25. Global Information Storage Capacity

Domain-driven design with its emphasis on dealing with complexity by breaking unwieldy monoliths into more manageable units in the form of subdomains and bounded contexts, fits naturally to this style of programming. Hence, it is no surprise to see a renewed interest in adopting DDD principles and techniques when crafting modern solutions. To quote Eric Evans, it is no surprise that Domain-Driven Design is even more relevant now than when it was originally conceived!

Summary

In this chapter we examined some common reasons for why software projects fail. We saw how inaccurate or misinterpreted requirements, architecture (or the lack thereof), excessive technical debt, etc. can get in the way of meeting business goals and success.

We looked at the basic building blocks of domain-driven design such as domains, subdomains, ubiquitous language, domain models, bounded contexts and context maps. We also examined why the principles and techniques of domain-driven design are still very much relevant in the modern age of microservices and serverless. You should now be able to appreciate the basic terms of DDD and understand why it is important in today's context.

In the next chapter we will take a closer look at the real-world mechanics of domain-driven design. We will delve deeper into the strategic and tactical design elements of DDD and look at how using these can help form the basis for better communication and create more robust designs.

Questions

1. What are the most common reasons for software projects to fail?
2. Why is DDD relevant in today's context?

Further Reading

Title	Author	Location
Pulse of the Profession - 2017	PMI	https://www.pmi.org/-/media/pmi/documents/public/pdf/learning/thought-leadership/pulse/pulse-of-the-profession-2017.pdf
Pulse of the Profession - 2020	PMI	https://www.pmi.org/learning/library/forging-future-focused-culture-11908
Project success: Definitions and Measurement Techniques	PMI	https://www.pmi.org/learning/library/project-success-definitions-measurement-techniques-5460
Project success: definitions and measurement techniques	JK Pinto, DP Slevin	https://www.pmi.org/learning/library/project-success-definitions-measurement-techniques-5460

Title	Author	Location
Analysis Paralysis	Ward Cunningham	https://proxy.c2.com/cgi/wiki?AnalysisParalysis
Big Design Upfront	Ward Cunningham	https://wiki.c2.com/?BigDesignUpFront
Enterprise Modeling Anti-Patterns	Scott W. Ambler	http://agilemodeling.com/essays/enterpriseModelingAntiPatterns.htm
A Project Manager's Guide To 42 Agile Methodologies	Henny Portman	https://thedigitalprojectmanager.com/agile-methodologies
Domain-Driven Design Even More Relevant Now	Eric Evans	https://www.youtube.com/watch?v=kIKwPNKXaLU
Introducing Deliberate Discovery	Dan North	https://dannorth.net/2010/08/30/introducing-deliberate-discovery/
No Silver Bullet — Essence and Accident in Software Engineering	Fred Brooks	http://faculty.salisbury.edu/~xswang/Research/Papers/SERelated/no-silver-bullet.pdf
Mastering Non-Functional Requirements	Sameer Paradkar	https://www.packtpub.com/product/mastering-non-functional-requirements/9781788299237
Big Ball Of Mud	Brian Foote & Joseph Yoder	http://www.laputan.org/mud/
The Forgotten Layer of the Test Automation Pyramid	Mike Cohn	https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid
Tech debt: Reclaiming tech equity	Vishal Dalal et al	https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/tech-debt-reclaiming-tech-equity
Is High Quality Software Worth the Cost	Martin Fowler	https://martinfowler.com/articles/is-quality-worth-cost.html#WeAreUsedToATrade-offBetweenQualityAndCost

[1] https://www.thoughtworks.com/en-us/radar/techniques/inverse-conway-maneuver

[2] https://wiki.c2.com/?PrimitiveObsession

[3] https://martinfowler.com/eaaCatalog/repository.html

[4] https://martinfowler.com/bliki/AnemicDomainModel.html