Table of Contents

The Rationale for Domain-Driven Design . . . 1
- Introduction . . . 1
- Why do software projects fail? . . . 2
  - Inaccurate requirements . . . 2
  - Too much architecture . . . 3
  - Too little architecture . . . 3
  - Excessive incidental complexity . . . 3
  - Uncontrolled technical debt . . . 4
  - Ignoring Non-Functional Requirements (NFRs) . . . 5
- Modern systems and dealing with complexity . . . 6
  - How software gets built . . . 6
  - Complexity is inevitable . . . 7
  - Optimizing the feedback loop . . . 9
- What is Domain-Driven Design? . . . 10
- Why is DDD Relevant? Why Now? . . . 11
  - Rise of Open Source . . . 12
  - Advances in Technology . . . 13
  - Rise of Distributed Computing . . . 13
- Summary . . . 14
- Questions . . . 14
- Further Reading . . . 14

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.

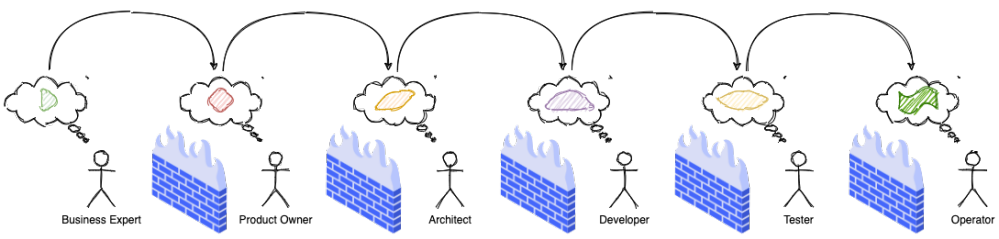

Figure 1- 1. Silo mentality and the loss of information fidelity

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:

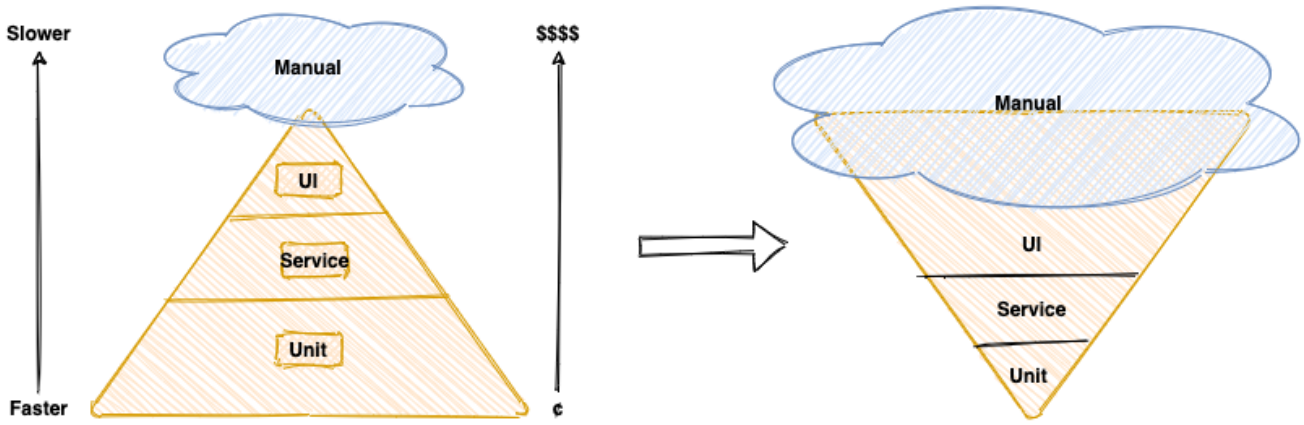

Figure 1- 2. Testing Strategy: Expectation vs. Reality

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:

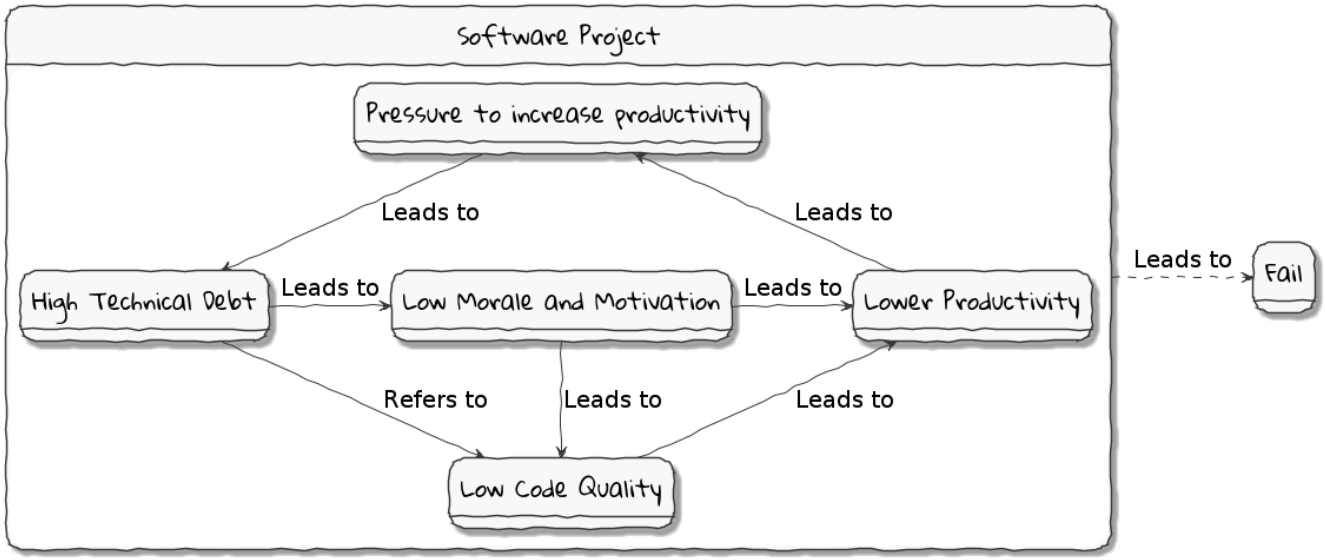

Figure 1- 3. Technical Debt — Implications

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:

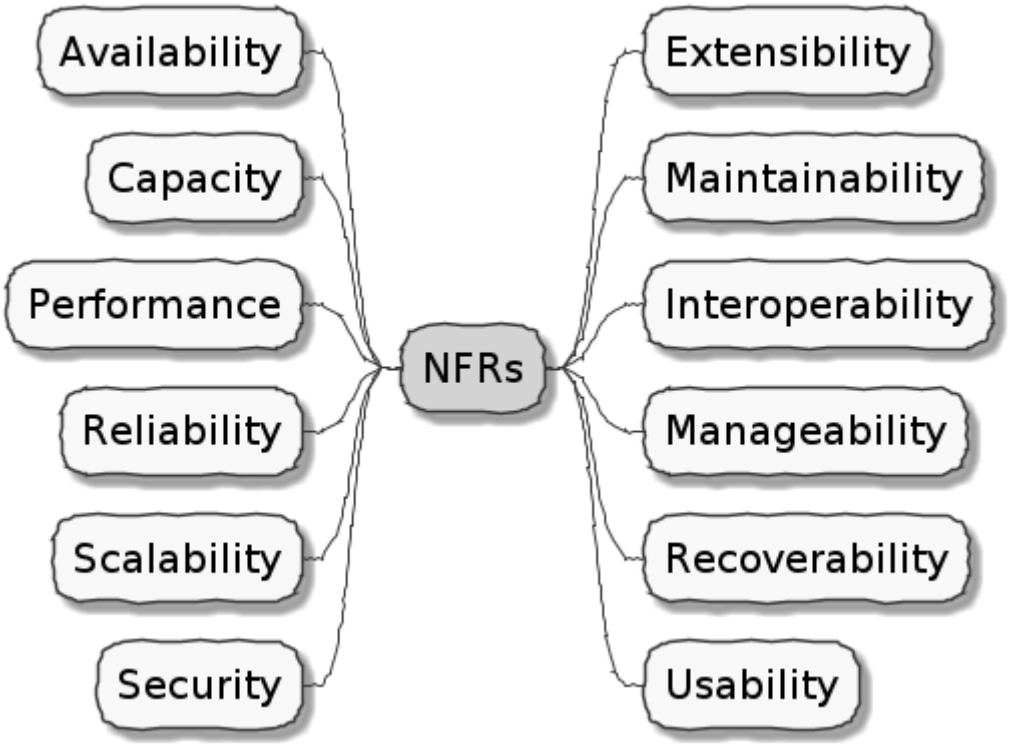

Figure 1- 4. Non-Functional Requirements

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:

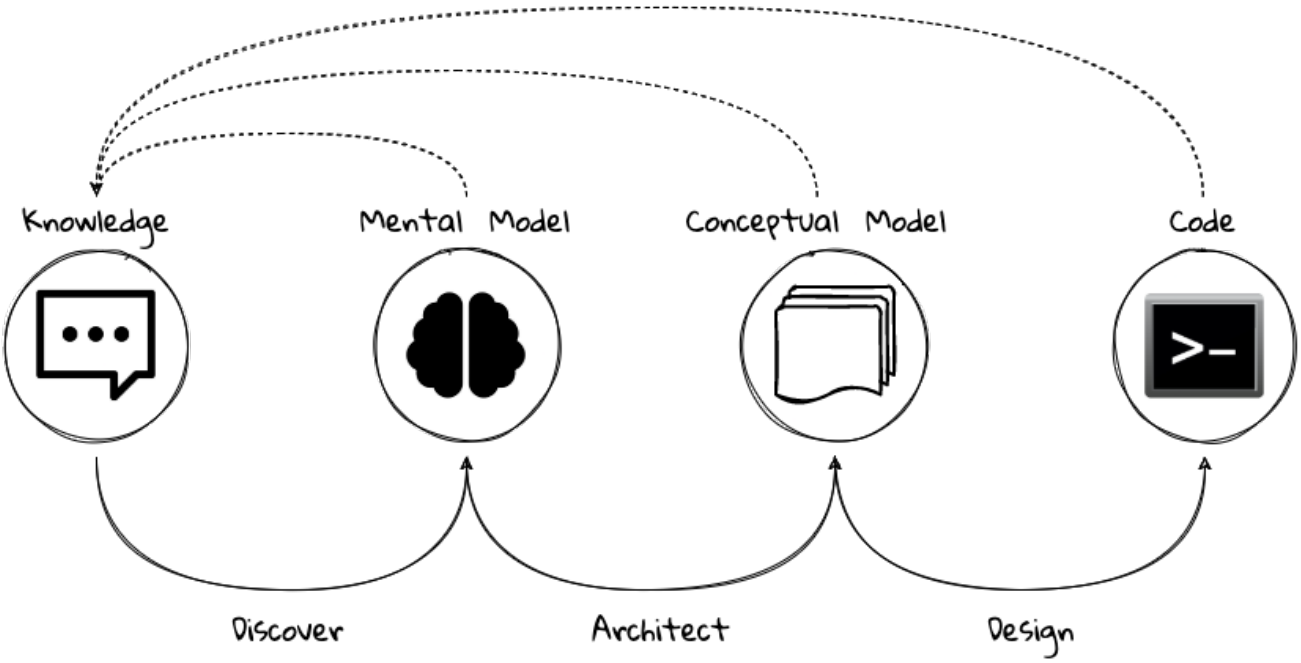

Figure 1- 5. Building software is a continuous refinement of knowledge and models

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,

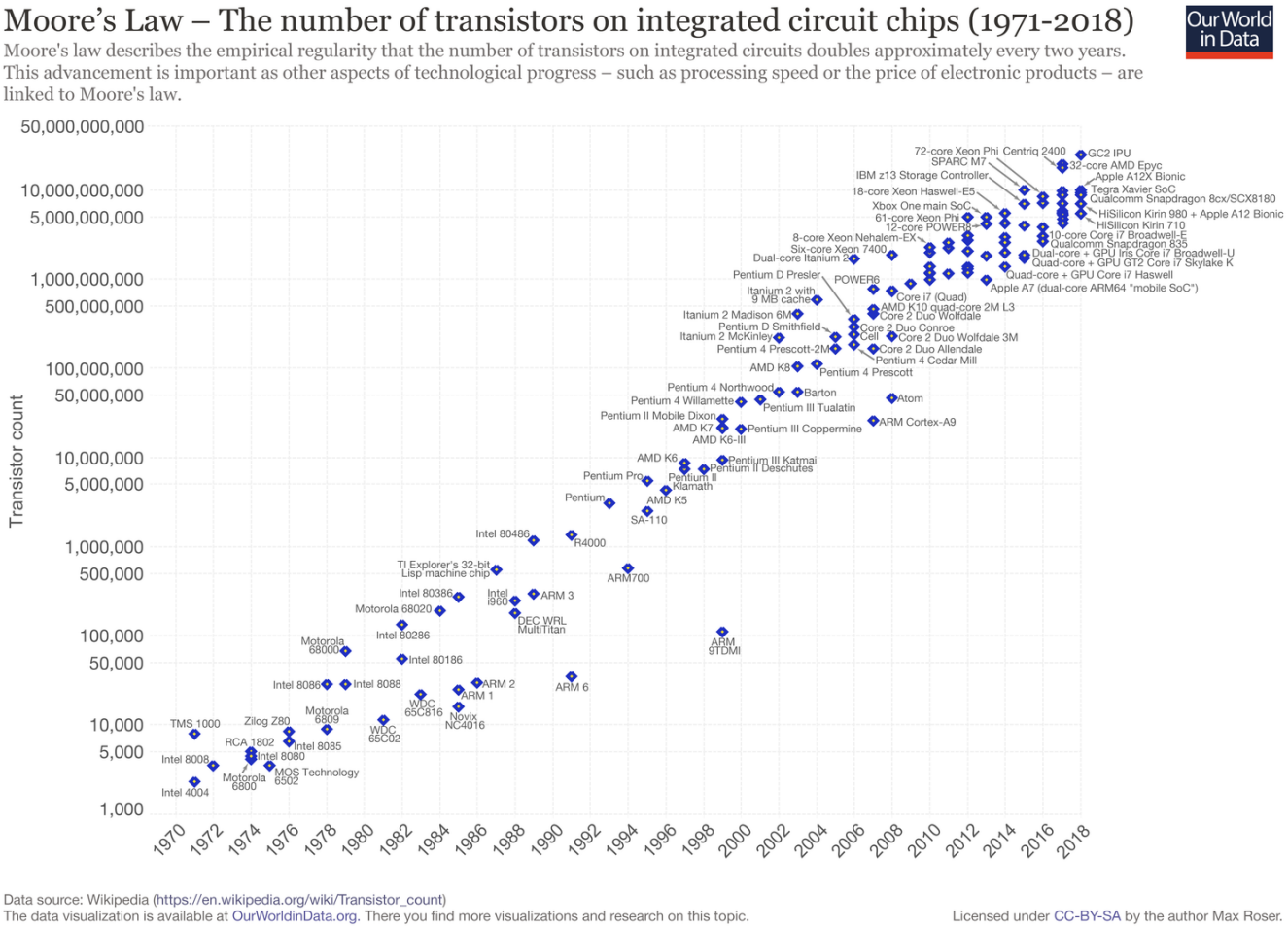

Figure 1- 6. Moore's Law

along with the rise of the internet as illustrated below,

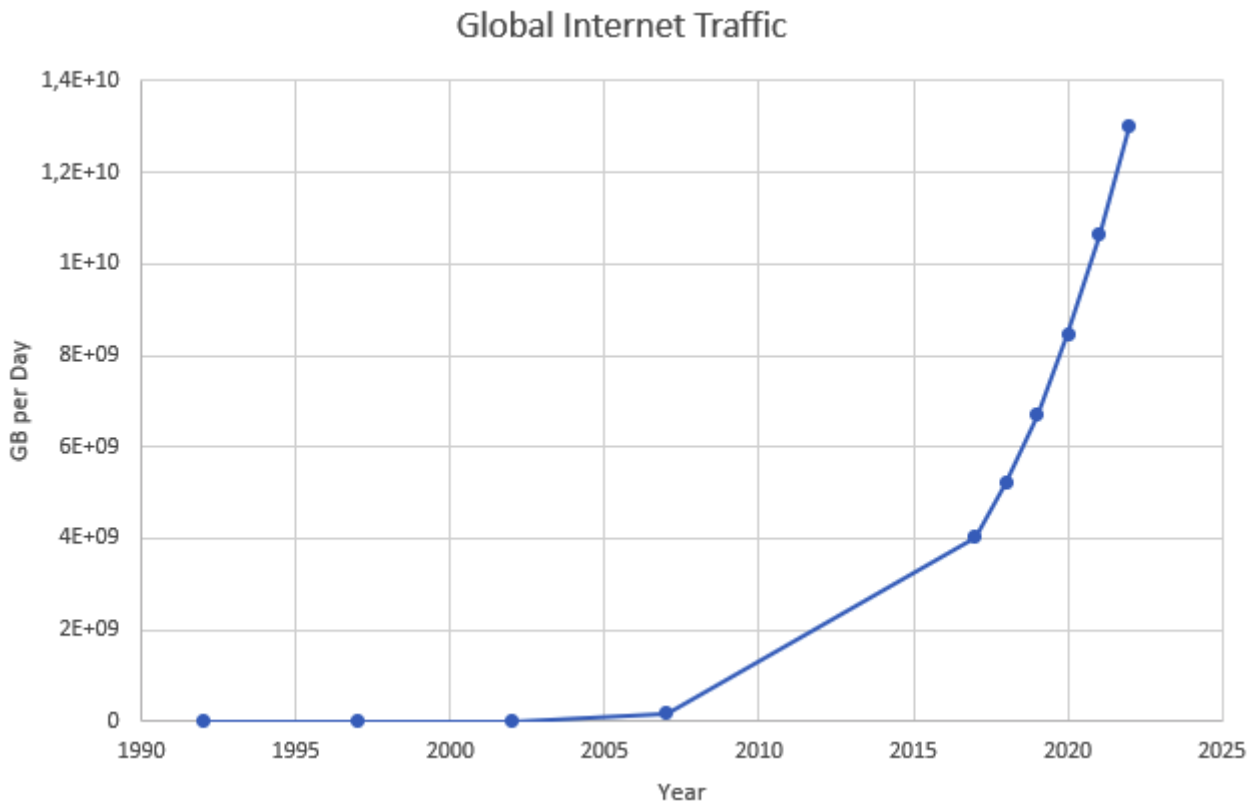

Figure 1- 7. Global Internet Traffic

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:

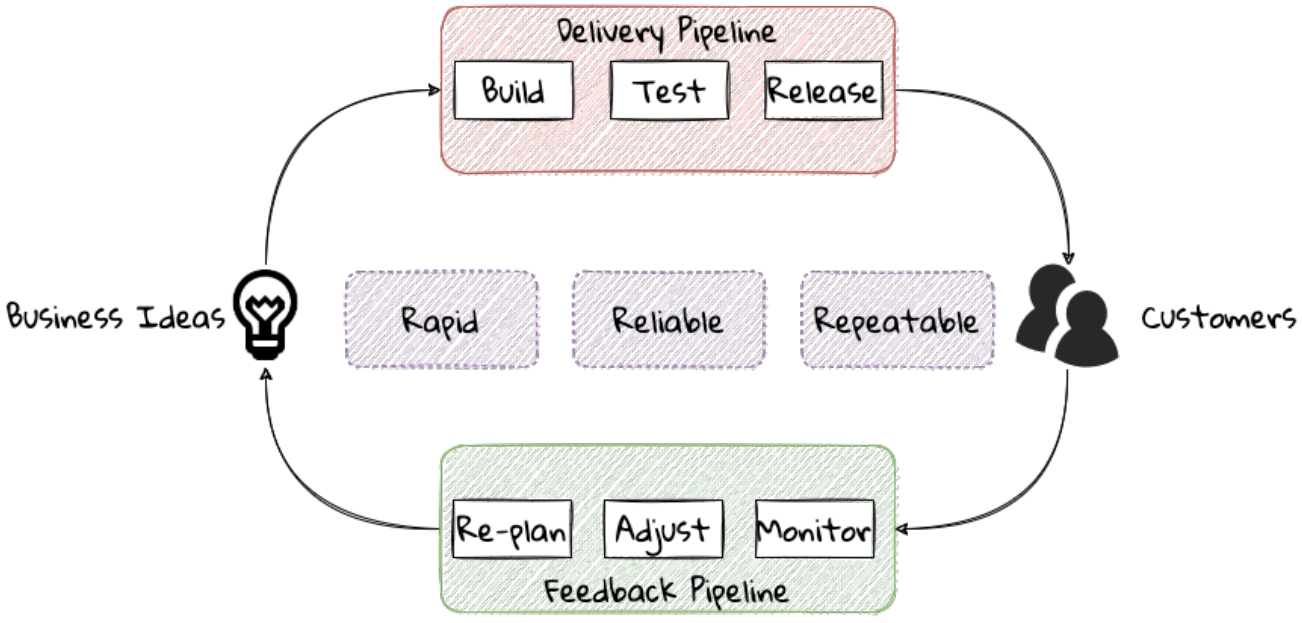

Figure 1- 8. The software delivery continuum

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:

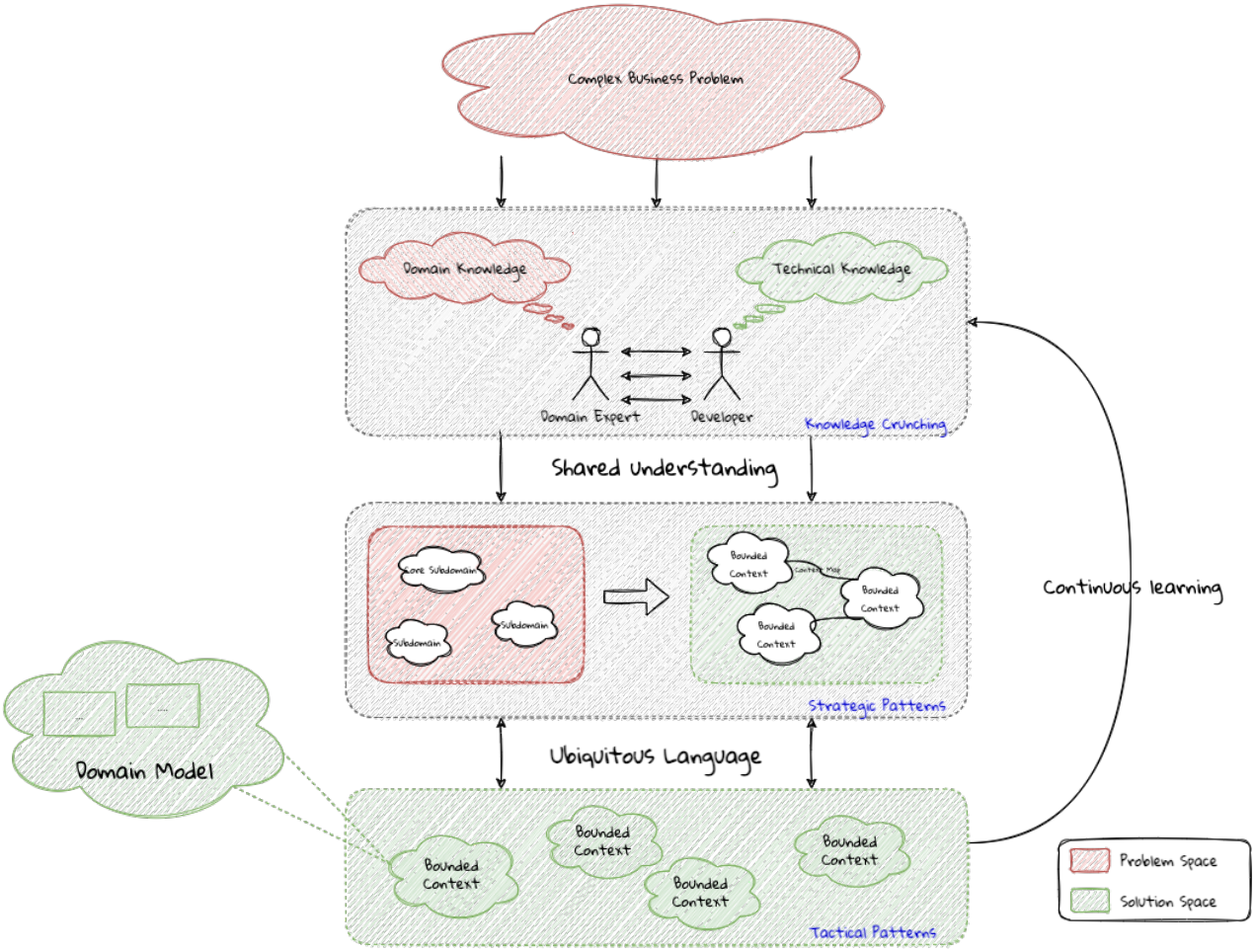

Figure 1- 9. Essence of DDD

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

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:

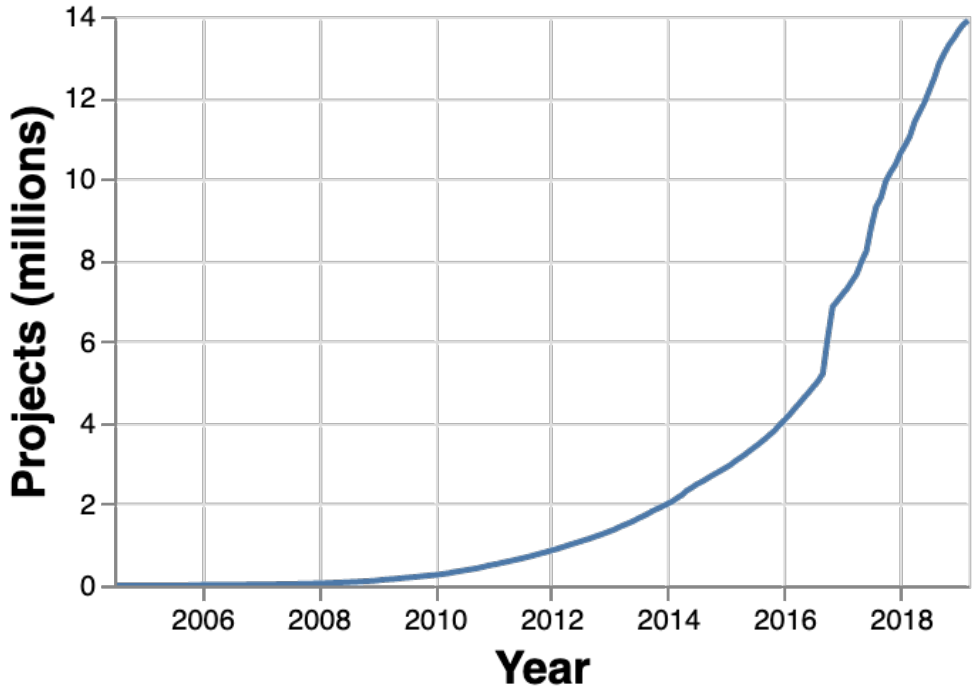

Figure 1- 10. Open source Java over the years. Source: https://mvnrepository.com/

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- 11. 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