O'REILLY®

Designing Autonomous Teams and Services

Deliver Continuous Business Value through Organizational Alignment

Nick Tune & Scott Millett

9 781491 994313

Designing Autonomous Teams and Services

Deliver Continuous Business Value through Organizational Alignment

Nick Tune and Scott Millett

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Designing Autonomous Teams and Services
by Nick Tune and Scott Millett

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Brian Foster and Jeff Bleiel
Production Editor: Justin Billing
Copyeditor: Jasmine Kwityn
Proofreader: Charles Roumeliotis

Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

September 2017: First Edition

Revision History for the First Edition
2017-09-05: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491994313* for release details.

978-1-491-99431-3

[LSI]

Table of Contents

CHAPTER 1

High-Performance Teams Continuously Deliver Business Value

We want our teams and services to be tightly aligned, but loosely coupled.

—Dianne Marsh, *Director of Engineering, Cloud Tools, at Netflix*

Modern, high-performing organizations employ continuous discovery and delivery to develop better products faster than their competitors. They are constantly running experiments to discover innovative new ways to solve customer problems, and they build high-speed engineering capabilities to deliver value every day, creating ultra-short customer feedback cycles. As Table 1-1 shows, the Puppet 2017 State of DevOps Report found high-performance organizations deploy to production 46 times more frequently than low performers, with lead times for changes an eye-watering 440 times faster.

Table 1-1. High IT performers deploy more frequently (source: Puppet 2017 State of DevOps Report)

	High IT performers	Low IT performers	Difference
Deployment frequency	Multiple times per day	Between once per week and once per month	46x more frequent
Lead time for changes	Less than one hour	Between one week and one month	440x faster

Organizations turn to agile out of their need to create better products faster. In fact, the 11th Annual VersionOne State of Agile Report found the number one reason for agile adoption was accelerated product delivery, followed closely by the need to improve business/IT alignment. This report will show you how the two topics of accelerated product delivery and improved business/IT alignment are inextricable. You will learn the techniques leading organizations use to improve alignment and accelerate product delivery by increasing autonomy (Figure 1-1).

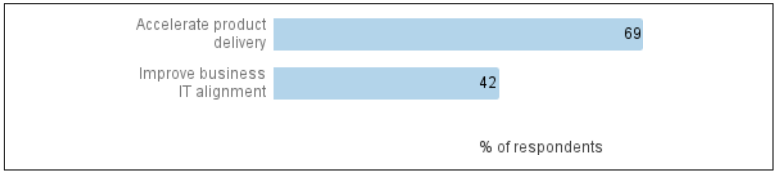

Figure 1-1. Top reasons for agile adoption (source: the VersionOne 11th State of Agile Report)

Low Autonomy Precludes Continuous Business Value

Organizations struggle to accelerate product development due to ineffectual alignment between business goals, team boundaries, and software architecture. Unnecessary dependencies are created (both within the system of the software as well as among the teams responsible for maintaining it), precluding the autonomy needed for teams to move fast and innovate. Teams do not have the autonomy to own problems from end to end—from interacting with customers to delivering engaging digital products.

Teams that do have end-to-end accountability are motivated by their freedom to creatively find the best solutions for problems with the highest business value. They can see the value of their work and they are part of the big picture. In Chapter 2, you'll see how to create a culture of knowledge dissemination that empowers all employees with knowledge of which business problems are the most important. As we continue in later chapters, you'll see how to align organizational and technical boundaries so teams can take full ownership of specific problems from end to end. The fundamental importance of autonomy cannot be understated. As psychologist and best-selling author Dan Pink argues:

We have three innate psychological needs—competence, autonomy, and relatedness. When those needs are satisfied, we're motivated, productive, and happy.

Low Autonomy Derails Digital Transformation in the UK Government

Since 2012, the UK government has been on an agile and digital transformation of unprecedented scale. Led by a center of excellence known as Government Digital Service (GDS), the project's ambition is for all technology departments in the UK government to be user-centric and highly agile. The need for the formation of GDS was clear after the UK government wasted £12 billion on "the biggest IT failure ever seen".

In order to help government IT departments who were rooted in years of heavy waterfall software development and big IT outsourcing contracts, GDS created service standards, educating teams how to focus on user needs by encouraging continuous user research. The standards also educate teams on how to be agile by working in smaller iterations and frequently deploying improvements to services.

Despite the extensive support from GDS and executive backing from senior government officials, some government departments were struggling with the transition. One government agency faltered immensely during their digital transformation—they weren't prepared for the far-reaching changes and higher levels of autonomy required to enable business agility.

Aversion to changing waterfall organization structure

A large part of the agency's struggles stemmed from their aversion to changing organization structure. They wanted to be agile, but they also wanted to maintain activity-oriented teams of specialists (frontend teams, backend teams, data teams, etc.). Understandably, they were fearful of making big changes.

Teams of specialists make continuous delivery almost impossible to achieve due to high handover costs. When work flows from frontend to backend teams, they must collaborate on how software will integrate and how the work will be scheduled, involving lots of meetings before the work is done and lots of quality assurance after.

Because handover costs are so high, work is done in larger batches to minimize handover costs.

As a consequence of larger batches, lead times grow, especially when each team has its own backlog, priorities, and is measured on the quantity of work completed rather than business value delivered. Bigger batches of work result in more context switching due to the amount of active work in the system, so less time is spent adding value. However, for organizations needing to innovate, the biggest danger is extended customer feedback cycles caused by longer lead times. Extended customer feedback cycles hold incorrect assumptions in the system, negatively influencing product and technology decisions for a longer period of time.

Bottleneck teams break business agility

In the government agency, only frontend teams working on websites were truly agile, deploying software to production on a daily basis. Whenever business rule changes or database changes were required, it would take months as frontend teams were limited by the lead times of backend teams they depended on. The backend teams were still deeply entrenched in the waterfall mode of operation despite practicing agile rituals. Senior management saw backend teams doing standups and Kanban boards and thought they were doing textbook agile. They couldn't see how the backend teams were bottlenecks, slowing the entire agency's rate of development (see Figure 1-2).

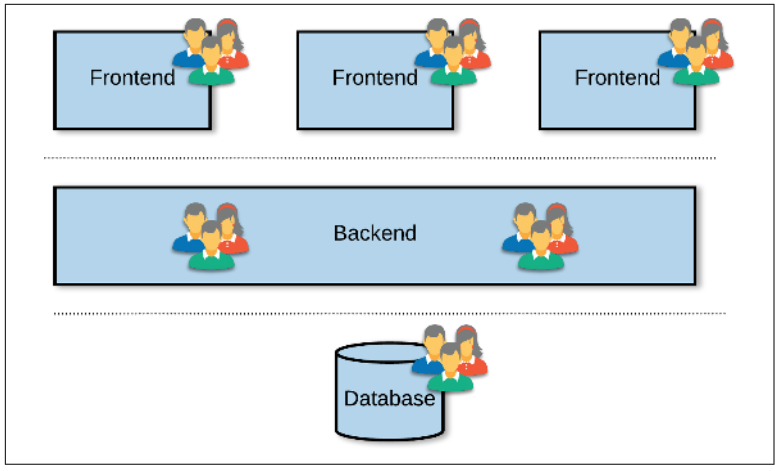

Figure 1-2. Waterfall organization and technical architecture

During the transformation, the agency was building a new service that was supposed to make it easier for citizens to declare important information related to their taxes. However, shortly after going live, a deluge of dissatisfied users were angrily complaining they could not register with the service. Citizens needed to use their address to register, but the address field on the web page was too short.

Increasing character limits required frontend, backend, and database changes. Frontend teams could make their changes and deliver updates to production in just a few hours. But the backend and data teams took months to implement changes. In addition, there were endless meetings between the teams as they tried to work out how the changes would interface. Would new APIs or integrations need to be built? When would the work be scheduled and prioritized across the many teams? Which team's project manager gets ultimate power?

Eventually, after many stressful meetings and much disenchantment, the work was put on hold indefinitely. It was too expensive, involving many changes across various teams who had many different initiatives in progress. It should have been a simple programming task, addressing serious user problems, yet the structure of the organization and software architecture made it too difficult to even attempt. This type of problem was common in the organization. The underlying reason was a lack of business agility—an inability to make rapid, iterative changes across the whole value stream.

The problem of half agile organizations with competing digital and enterprise silos was so prolific that shortly before he left his role as Head of GDS, Mike Bracken becried, "You can't be half agile." Referring to the same problem, his compatriot in the Australian Digital Transform Agency, Paul Shetler, made similar remarks shortly after leaving his post. He stated, "Digital transformation shouldn't just be lipstick on a pig."

Lack of ownership leads to blame culture

When teams must collaborate to deliver a single user-facing feature, there is an elevated risk of blame culture, as was present in the government agency. When projects overran, errors cropped up in production, or customers were complaining about bugs, teams were always trying to defend their work and ensure they weren't held

responsible, blaming other teams at the expense of resolving urgent user problems as quickly as possible.

On one occasion, a project was running late. New government legislation coming into effect dictated that the new service must be live by the agreed date. Teams were already starting to feel huge pressure and beginning to point the finger of blame. When the system finally did go live, the blame culture surged to unprecedented levels. Shortly after launch, response times plummeted and users were complaining loudly. It was a big embarrassment for senior management. Meanwhile, all of the teams were aggressively defending their parts of the system and condemning other teams. No team wanted to admit fault for the error, so no team felt obliged to fix it. The problem persisted far longer than it should have.

Blame culture was just the symptom of deeper systemic flaws. Low autonomy was the underlying cause of the problems. No team had the autonomy to own the end-to-end problem and fix it. A highly coupled organization resulted in a highly coupled software system with high maintenance and integration costs.

Technology alone does not confer agility

When the opportunity arose for the government agency to address its biggest problems and strive toward business agility, it made the mistake of trying to buy agility with expensive enterprise software. The traditional favorites were all thrown into the mix: the generic rules engine, the business process management (BPM) tool, and the enterprise service bus (ESB). See Figure 1-3.

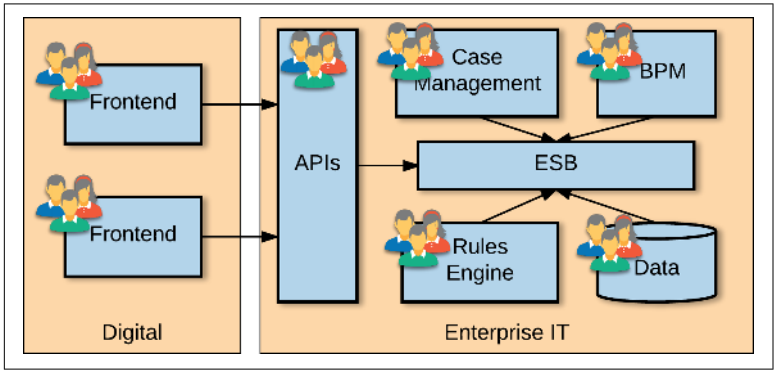

Figure 1-3. More tools led to more silos and bottlenecks

During an important strategy meeting, the CTO asserted: "Developers have been a liability to the business. They are too slow." Unfortunately for the agency, he identified the wrong solution. He continued, "the industry is moving towards these tools [generic rules engines, BPM tools] that don't need programmers," pointing at the consultant selling the tools as justification. The CTO had fallen for flashy enterprise IT marketing campaigns that tempt executives with the promise of digital transformation without having to make organizational and cultural changes.

After building the new enterprise platform, the proliferation of software components and teams resulted in painful coordination overheads and massive lead times. During a periodic progress report, one project manager painstakingly explained, "Coordinating all of these teams is such a huge challenge, and it's taking so much time, but we're starting to make some progress now," pointing to an illegible diagram highlighting the tangled web of dependencies between the many teams. The problems were all due to the poor alignment between organizational and technical boundaries, and could have been avoided.

Many organizations wishing to become agile succumb to the same fallacy of thinking they can simply buy tools to make them agile. However, as Martin Fowler, Chief Scientist at Thoughtworks articulates, these tools rarely deliver on their promise:

> *Often the central pitch for a rules engine is that it will allow the business people to specify the rules themselves, so they can build the rules without involving programmers. As so often, this can sound plausible but rarely works out in practice.*

User experience suffers from low accountability

With a fractured organization and technology estate, user experience is one of the biggest casualties. Poor performance, extensive lead times to fix simple bugs, and weird inconsistencies due to silo teams all worsen UX, and it's not just external users who suffer. During one project, a component was not completed in time for the deadline. Consequently, JSON HTTP requests were printed out on paper and manually typed into an internal case management system. One case worker justifiably groaned: "The new system was supposed

to save us time so we could clear the huge backlog of cases. But now work is taking even longer and the cases are piling up."

What Actually Is Business and Customer Value?

For most organizations the focus for a delivery team is to create business value (business value being subjective and based on the nature of a business). The primary reason a for-profit organization exists is to make money for its shareholders, so business value is equatable to the ability to make, protect, or save money. Importantly, business value is not just about maximizing short-term shareholder value.

For nonprofit organizations such as governmental departments or charities, business value can be measured as a particular task, such as curing a disease or decreasing the amount of people dependent on the welfare state.

Customer value is determined by the ability to increase customer satisfaction through products or services. Customer value is also heavily influenced by the context of an organization. In a for-profit organization, customer value is a means to increase business value. Customer value should be seen as a measure of how valuable a customer finds an organization's products or services. Focusing on giving value to the customer will lead to increased revenue, which results in business value.

High-Performance Teams Are Autonomous

Autonomy enables two essential characteristics of high-performance teams. Product strategy autonomy enables teams to work closely with customers, continuously discovering what is valuable to customers by conducting user research and experiments on an ongoing basis. Architectural autonomy enables teams to build high-speed engineering capability so they can deliver business value frequently with minimal coordination overhead.

These two characteristics form the modern version of agile—referred to as dual-track agile or continuous discovery and delivery, as shown in Figure 1-4. Many ambitious organizations around the world are rapidly adopting this approach to product and software development because it enables them to not only build better prod-

ucts for their customers with greater efficiencies, but to unearth innovative new product ideas and revenue streams.

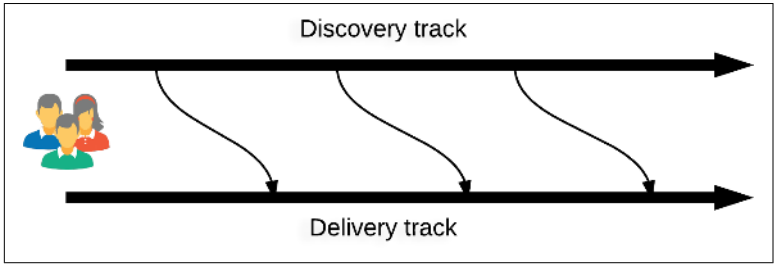

Figure 1-4. The dual tracks of continuous discovery and delivery

Teams practicing continuous discovery and delivery thrive in a world of constant change. They are poised to take advantage of new opportunities presented by technological advancements and changing consumer behaviors. However, the majority of organizations are still way behind.

In a poll of Fortune 500 CEOs conducted by Fortune.com, the rapid pace of technological change was identified as the single biggest challenge. Contrastingly, high-performance organizations relish the rapid pace of change. It is viewed as an opportunity to increase competitive advantage and create new revenue streams.

Amazon is widely regarded as one of the world's most innovative companies. Continuous discovery is at the heart of their culture as Jeff Bezos frequently articulates:

> *Companies that don't embrace failure and continue to experiment eventually get in the desperate position where the only thing they can do is make a Hail Mary bet at the end of their corporate existence.*

Autonomy to Continuously Discover User Needs

Continuous discovery is the process of continuously running experiments and talking to customers in parallel with delivering new features and product enhancements. Autonomous teams are empowered to directly engage with customers and shape their own product roadmaps. Teams continually deepen their awareness of customer needs and pain points, instead of just executing on a plan dictated by management. Innovation happens from the ground up and people are thoroughly inspired by their work.

Such is the importance of continuous discovery: the Alpha UX 2017 Product Management Insights survey concluded the biggest source of product and feature ideas was direct customer feedback (see Figure 1-5). Teams practicing continuous discovery are, therefore, exploiting the full potential of the biggest source of product ideas.

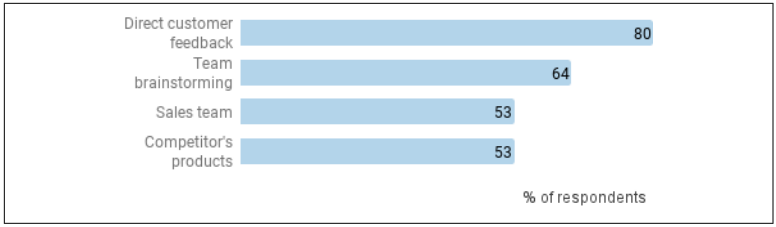

Figure 1-5. Biggest sources of product and feature ideas (source: Alpha UX 2017 Product Management Insights)

How continuous discovery works

Many activities can be applied as part of a continuous discovery strategy, both qualitative and quantitative, involving talking to customers or measuring their behavior. Some of the most popular and effective discovery activities are lab sessions (in a user research lab, not a science lab), visiting users at their home or place of work, online surveys, web traffic analysis, A/B testing, and website feedback links. Discovery should not be carried out by a separate team; delivery teams must run their own discovery track.

Continuous discovery at Coop Digital

Coop Digital is one of the UK's leading agile tech organizations, putting user needs at the heart of everything they do. They are continuous discovery experts. Coop have shared a powerful and evocative example demonstrating the benefits of continuous discovery on their blog; their case study articulates how Coop dramatically improved the quality of charity donations from their members.

Coop is a cooperative (i.e., a business owned by its members rather than investors). Coop is also an ethics-driven business—use of fair-trade products, support for local farming, and tackling climate change are among their biggest core values. It's these values that attract members to joining the Coop. Surprisingly, though, users weren't taking advantage of the ability to influence where their charitable donations were sent. This didn't align with their core values, as Simon Hurst explains on the Coop blog. Something wasn't right.

With their approach to continuous discovery, Coop were regularly unearthing UX problems by running lab sessions with users, getting whole delivery teams involved. During lab sessions, users were telling Coop they did not even realize they had the ability to choose which local causes their donations could be shared with, even though there was a link on the membership page allowing them to do so. This discovery was also backed up with data, as Simon explains: "We saw that a significant number of people were getting to the page with the 'call to action' (the bit where they could choose a cause) but they weren't actually selecting one."

To resolve the issue, rather than rush to implement a solution, Coop continued in discovery mode, prototyping new designs and testing them with users in the lab. Having discovered the user need and the solution that performed best with research participants, Coop promoted the solution from their discovery track to their delivery track. A production-quality solution was engineered and released for all users, resulting in an immediate 10% increase in usage.

In the article, Hurst also reinforces the need for the full team delivering the product to be involved in discovery. He explains that:

> *Whilst we're responsible for user research, the whole team get involved in research sessions and meeting users so they can empathise with the people who use the services we're building. This ensures they design with the user, and not themselves, in mind.*

Autonomy to Continuously Deliver Business Value

Having the autonomy to continuously deliver business value enables high-performance teams to maximize the rate at which customers receive value. Once hypotheses have been tested on small numbers of users on the discovery track, they are promoted to the delivery track where teams will speedily deliver features and validate them on larger numbers of users. Continuous delivery is, therefore, immensely important for organizations wishing to accelerate product development. To take full advantage of accelerated product development and more reliable software delivery, teams practicing continuously delivery deploy to production every day.

Continuous delivery confers other advantages, enabling faster and more reliable software delivery. It reduces risk by delivering smaller product increments. The amount of work being released is smaller

so chances of failures are smaller. When problems do happen, they are easier to identify and recover from.

How continuous delivery works

Engineering teams practicing continuous delivery work in small batches. As soon as they complete a work item it is deployed to production for users to benefit from, in contrast to traditional approaches where work is batched up and delivered at regular periods. As a consequence, achieving continuous delivery is not easy. It requires a deep investment in engineering culture and practices. With large batch sizes, manual regression testing can be performed on the entire batch to verify there are no defects. With continuous delivery, there is no time to run manual regression tests before every release when you are deploying multiple times per day. For example, Alistair Hann of SkyScanner Engineering claims, "We may get to 10,000 releases per day at the end of next year [2017]."

To accommodate continuous delivery, software developers have to adopt practices that enable them to build quality into their software so that extensive manual testing is not needed prior to every release. To build quality into the product, agile software development teams will utilize practices such as test-driven development and pair programming to increase the robustness and confidence in the code. Equally, continuous delivery necessitates highly automated infrastructure in addition to code build and deployment pipelines to ensure the cost of deploying is negligible. Robust metrics and monitoring must be in place so teams find out before their customers if problems creep into products. To learn more, Jez Humble and Dave Farley's *Continuous Delivery* (Addison-Wesley, 2010) is the bible.

Continuous Delivery Alone Does Not Enable Business Agility

Continuous delivery is essential yet, alone, is insufficient to enable business agility. So long as dependencies exist between teams necessitating high collaboration, the collaboration costs will slow down delivery, and the slowest moving team will act as a bottleneck, limiting the overall throughput of the other teams. See Figure 1-6.

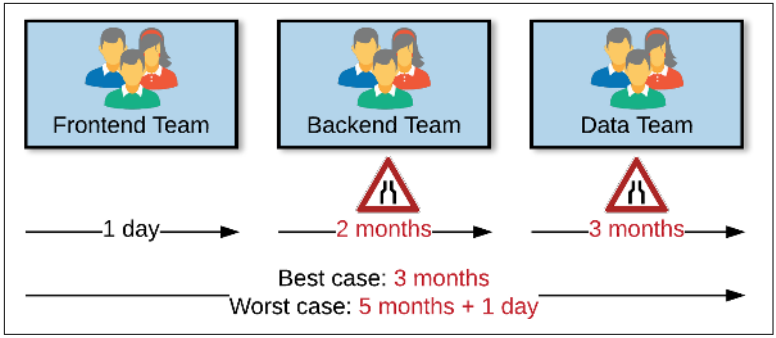

Figure 1-6. Bottleneck teams constrain the entire value chain

Many organizations have end-to-end lead times of one day, not just on their websites, but for any change, even one that involves complicated frontend, backend, and database changes. As more and more organizations apply modern best practices and achieve one-day end-to-end lead times, those that do not keep up will disappear.

Aligning Organizational and Technical Boundaries Enables Autonomy

To maximize autonomy and enable business agility, it is essential to align organizational and technical boundaries. Delivery teams must be granted authority of specific product capabilities, and must own all of the technical components needed to deliver their capabilities, as supported by findings from the Puppet 2017 State of DevOps Report:

> *Loosely coupled architectures and teams are the strongest predictor of continuous delivery. If you want to achieve higher IT performance, start shifting to loosely coupled services—services that can be developed and released independently of each other—and loosely coupled teams, which are empowered to make changes.*

Loosely coupled teams aligned with business capabilities will develop a deep understanding of what is valuable to the business

and to customers. And teams will be primed for innovation, with few dependencies to get in their way.

McKinsey Global Survey Highlights Need for Autonomy in Digital Programs

Three of the five most significant challenges meeting priorities for digital programs were related to low autonomy and poor organization design, according to the McKinsey Global Survey: organization structure not appropriately designed for digital, business processes too inflexible to take advantage of new opportunities, and inability to adopt an experimentation mindset. In contrast, companies comprised of outcome-oriented autonomous teams aligned to business capabilities, with the ability to run experiments and shape their own roadmap, are the blueprint for modern digital enterprises.

Achieving high alignment between business goals, team boundaries, and software architecture is a challenging endeavor. You cannot expect an overnight transformation dictated by the CIO to lead to instant success. Transformation is iterative, requiring engagement and input from all levels of an organization. In fact, there is never a clear end state. Adaptability is a necessity—new business strategies or shifts in priorities often need realigned boundaries to flourish.

Through a culture of knowledge dissemination, cross-functional design, and strategy alignment, your organization will be capable of continuously adapting boundaries to better deliver business value. In the famous words of Spotify's Henrik Kniberg, "Alignment enables autonomy." When teams are aligned on strategy, they have the situational awareness to autonomously make good decisions in the best interests of the company. This report will demonstrate the cultural principles and collaborative practices used by high-performance organizations to enable aligned autonomy. See Figure 1-7.

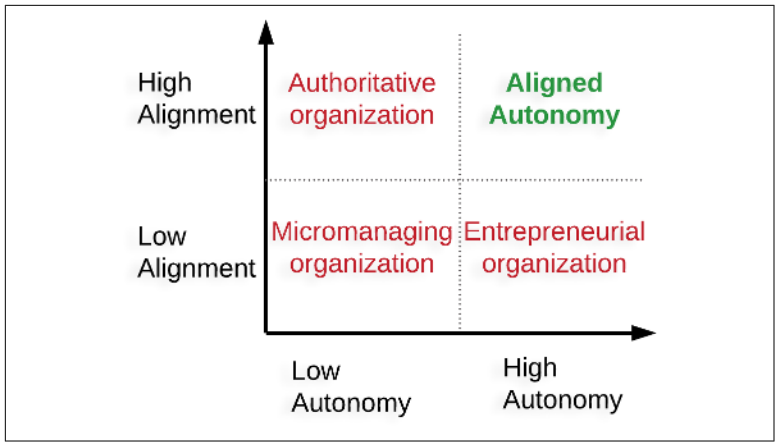

Figure 1-7. Aligned autonomy (source: Henrik Kniberg)

Aligning Boundaries Maximizes Continuous Discovery and Delivery

Finding things that change together for business reasons is the key to aligning organizational and technical boundaries, maximizing the amount of a team's work that is not dependent on other teams.

Aligning boundaries maximizes customer responsiveness (i.e., the speed at which you can satisfy the needs of your customers) because a single team will own the work from start to finish and won't incur the expensive costs of collaborating with multiple teams with different priorities and backlogs (Figure 1-8). Entire value chains are streamlined reducing end-to-end lead times and accelerating product development. Higher autonomy also leads to greater efficiency, as teams spend more time on creating value rather than context switching or coordinating.

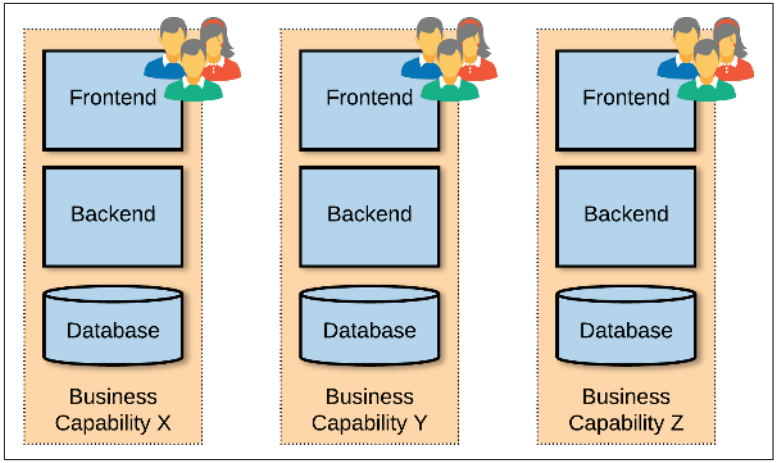

Figure 1-8. Vertically sliced product teams

Align boundaries to maximize customer responsiveness

When the previously mentioned government agency wanted to increase the character limits on a web form to fix an urgent bug, they hit a roadblock trying to coordinate their various teams. It took them weeks of effort, and still they could not satisfy the needs of their users.

If the agency had been organized into autonomous teams aligned with business capabilities, a single team would have owned the end-to-end capability and been able to resolve the issue in just a few hours. The team would have owned frontend, backend, and data for the capability. They wouldn't have required any meetings with other teams. No sequencing of work and juggling backlogs of multiple teams would have been necessary. The challenge of managing changes to technical integration points across different teams would not have existed. Aligning organizational and technical boundaries with a specific business capability would have eliminated the collaborative overhead and dramatically improved customer responsiveness and efficiency in the agency.

Align boundaries to potentiate discovery

Aligning organizational and technical boundaries increases the effectiveness of continuous discovery. When ideas are promoted from discovery to delivery, there will be no bottlenecks. The people who were involved in discovery will be implementing the solution

end to end. They will understand the value of their work and be emotionally invested in its success. They will be able to prioritize accordingly and deliver the solution which they decide is the most valuable option in the backlog. They will not be blocked waiting for other teams.

Not just vertical slices: Theory of Constraints

One final point to be aware of is the misconception that simply moving to vertically sliced product teams is sufficient to achieve high performance. This is a dangerous fallacy that can actually create more problems than it solves. Vertical slices in isolation do not guarantee autonomy; there still may exist dependencies between vertical slices resulting in bottlenecks.

Consider the following three-stage government process: Review, Resubmit, and Renegotiate. During Review, businesses can view the information held about them used to calculate their taxes. Resubmit enables businesses to provide corrections to invalid data, and Renegotiate enables them to challenge how much tax they pay if any data has been proved to be incorrect. Striving for business agility, the government department creates autonomous Review, Resubmit, and Renegotiate teams, alongside an autonomous Case Management team. Teams are perceived to be autonomous, practicing continuous discovery and delivery, but still there are big problems.

As the Review, Resubmit, and Renegotiate teams discover new opportunities to create value, they regularly discover they need to start capturing additional data from users. Before they can deliver these new improvements, though, they have to wait for the Case Management team to update their system. Whatever information is supplied by users must be carried through to the Case Management system verbatim by law. Consequently, the Case Management team becomes a bottleneck. See Figure 1-9.

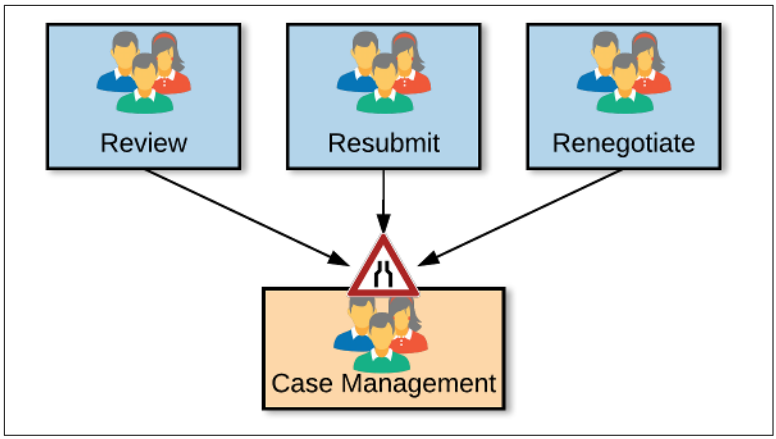

Figure 1-9. Bottlenecks can still exist between vertical slices

Instead of creating a dedicated Case Management team, the government agency could have more closely inspected things that change together for business reasons and chosen more suitable boundaries. For example, whenever there was a change to data formats in the Review, Resubmit, and Renegotiate teams, there was always a corresponding change required in the Case Management system—clear signs of a dependency.

To remove the dependency, the Case Management system could have been decomposed. Each part of the Case Management system could have been devolved into the context that it was coupled to, eliminating the bottleneck and giving each team the autonomy to improve end-to-end lead times, thus enabling business agility. See Figure 1-10.

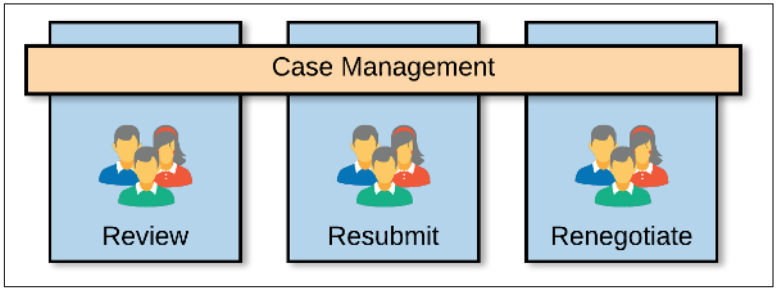

Figure 1-10. Eliminating bottlenecks with composite user journeys

In Summary: Enabling Teams to Be Autonomous

As you have read, teams with a high level of autonomy are able to deliver greater business value more frequently when they have:

- Aligned autonomy—that is, a shared understanding of the company's strategic context and key objectives
- Autonomy to make business and product decisions
- Autonomy to develop a rich understanding of user needs through continuous discovery directly with customers
- Autonomy to employ technical practices that enable continuous delivery
- Autonomy to organize technical and team boundaries around business outcomes

In the following chapters you'll learn how to achieve these characteristics and enable autonomous teams in your organization. Here's a brief snapshot of what we'll cover in each of the following chapters:

- Chapter 2, *Communicating the Business Context*, shows how to disseminate business context so teams are aligned to business goals, enabling them to be autonomous.
- Chapter 3, *Analyzing Domains*, explains how to analyze problem domains for conceptual cohesion, which is needed to align teams and architecture with closely related business concepts.
- Chapter 4, *Discovering Contexts*, teaches hands-on collaborative practice for modeling autonomy in problem domains, driven by core business and customer needs.
- Chapter 5, *Designing Antifragile Systems*, demonstrates how to analyze, explore, and optimize systems as a whole to avoid silos forming in the organization.
- Chapter 6, *Architecting Autonomous Applications*, introduces the most successful architectural patterns that enable teams to be autonomous.

CHAPTER 2
Communicating the Business Context

> *Leadership requires two things: a vision of the world that does not yet exist and the ability to create it.*
>
> —Simon Sinek

High-performance organizations are purpose-driven. They have clear, ambitious goals, and they make strategic and tactical decisions aligned with their goals. Leaders convincingly and repeatedly share their vision, while empowering teams to self-organize and find the best way to make the vision a reality.

Google is one of the biggest proponents of disseminating business context as a way to align individual and team goals with business objectives. Since 1999, Google has used OKRs (Objectives and Key Results),[1] which provide two primary benefits. First, anyone in the organization can see the current and previous goals of anyone else. A graduate software engineer working at Google could see the priorities of the CEO, for example. Second, OKRs align the goals of all workers with the strategic goals of the business via a cascade down the org chart.

Following Google's lead, many other high-performance organizations have adopted OKRs, including LinkedIn, Intel, and Twitter. But other approaches are also promoted by leading enterprises.

1 See "How Google Sets Goals: OKRs" for more information.

Salesforce uses a similar cascading approach known as V2MOM (Vision, Values, Methods, Obstacles, Measures).[2]

Spotify uses an approach called the Big Bets Spreadsheet, which gives all employees access to a prioritized list of business initiatives with links to further details, including teams contributing to them.[3]

In this chapter, you'll learn more about knowledge dissemination and alignment practices like OKRs. You'll also learn about more fine-grained approaches for delving into specific areas, including the Business Model Canvas and Impact Mapping. Importantly, you'll see how these techniques are useful not just for disseminating knowledge, but for engaging the whole organization in collaborative strategic decision making and innovation—an essential characteristic of innovative teams. No matter your job title, you can champion the use of these techniques to improve knowledge dissemination.

Proving Poor Alignment with a Sneaky Experiment

One person who has suffered the agony of dysfunctional organizations lacking business context is Madrid-based Software Development Manager Javier Fernández. At the DDD eXchange 2017 conference in London, Javier told the poignant story of how he once found himself utterly dismayed at the myriad problems plaguing the insurance company he worked for due to a fundamental disconnect between engineering teams and senior management. Javier was too conscientious to let the problems ruin the company he worked for. So he cooked up the most cunning of plans.

Javier independently ran workshops with the development team and the CEO in order to produce a Business Model Canvas. Later he asked each group to review the other group's canvas. Deviously, he didn't say who had created the canvas, leading each group to think it was his attempt at the canvas. Pulling off a masterstroke, he gained consent to record the sessions.

Fearing for his job, but desperate to transform his organization, Javier called a meeting and invited both groups. He began by showing

2 See "How to Create Alignment Within Your Company in Order to Succeed" for more information.

3 See "Alignment at Scale—Or How to Not Get Totally Unagile with Lots of Teams" for more information.

a clip of a Technical Lead unwittingly chastising the Business Model Canvas created by the CEO. After a few heated moments, Javier turned the tables by showing a clip of the CEO verbally ripping the development team's canvas to shreds.

Remarkably, after a few rounds of heated debate, Javier explains how both groups became empathetic, seeking to find a constructive path forward. From then on, Javier recounts, alignment and collaboration improved massively as they regularly began applying domain-driven design and collaborative practices. Javier's talk is highly recommended viewing. It is freely available online at *bit.ly/alignment-experiment*.

Context, Context, Context: Understand Your Business Context

As Salesforce CEO Marc Benioff claims, business context should drive all decision making:

> *V2MOM has been used to guide every decision at Salesforce.com —from those we made in 1999 to the decisions we make today as the largest high-tech employer in San Francisco.*

If decisions aren't aligned with the vision, how can they be in the best interests of the business? It sounds patronizingly obvious, yet many enterprises are too busy trying to copy fashionable trends rather than focusing on how to optimize for their own needs. There is immense hype around digital transformation, yet no one knows what it actually is.[4] Tech companies all over the world are rushing to copy the Spotify model, but there may not even be a Spotify model.[5]

As a consequence of not understanding their own business context, employees will unintentionally make poor strategic choices. In lieu of a clear sense of purpose, software developers will focus on shiny new technologies, product managers will just copy other products, and line managers will strive for 100% utilization instead of maximizing business value. You can go a long way to avoiding these

4 See "It's Time to Discard Digital" for an in-depth discussion.

5 See "There Is No Spotify Model" for more information.

problems by applying the following principles and practices to disseminate key business knowledge throughout your organization.

Cascading Objectives

Cascading objectives start at the very top of an organization. The CEO will internally publicize her vision and goals for the period (usually a quarter or year) and then direct reports will create their objectives based on the CEOs, cascading all the way down to front-line staff. Some approaches are purely focused on individuals, whereas some approaches have team- and department-level goals.

OKRs are the most prominent form of cascading objectives, used and publicly praised by Google, LinkedIn, Twitter, and other tech giants. Implementations of OKRs can vary. According to Rick Klau of Google Ventures, Google's approach to OKRs is based on the following:

- Objectives are ambitious, and should feel somewhat uncomfortable.
- OKRs are public; everyone in the company should be able to see what everyone else is working on (and how they did in the past).
- Key results are measurable; they should be easy to grade with a number (Google use a 0–1.0 scale to grade each key result at the end of a quarter).
- OKRs are not synonymous with employee evaluations. OKRs are about the company's goals and how each employee contributes to those goals.

The following OKR was set for himself by the CTO of an online travel company in Q3 2013:

Objective: Develop products faster than our competitors

Key Results:

- Each team deploying to production at least 3 times per week
- Cycle time less than 2 days for at least 90% of work items
- All engineers and testers to watch at least 1 user research session per month

Cascading objectives are susceptible to a few limitations. They can be hard to follow rigidly in times of uncertainty, for example. However, unless you are already applying a more effective technique, you should seriously consider adopting OKRs, V2MOM, or something similar.

Emphasize the Full Business Portfolio

Successful organizations always have a range of bets in play. They have a portfolio of products and innovations each at different phases of maturity. Having a broad view of the portfolio enables studious sharing of effort between what is valuable to the business now and what could bring value in the future. Try to avoid slipping into the insular mindset of encouraging a culture where people are focused on maximizing existing revenue streams to make the next quarter's figures look good. It shouldn't need repeating that too much focus on maximizing existing revenue streams and not enough investing in future innovations is a precarious strategy vulnerable to new innovations taken advantage of by competitors.

Consider using the Explore, Exploit, Sustain, Retire framework to ensure your portfolio contains a healthy mix of initiatives that serve your organization well in the short, medium, and long term. And importantly, use the framework to clearly communicate throughout your organization which initiatives are valuable, even if they aren't flagship offerings with huge revenues. People can then self-organize to invest their creative energies for maximum business impact.

Initiatives in the *Explore* phase are new areas of innovation for the business intended to create future revenue streams. Initiatives in the *Exploit* phase have been validated with customer feedback and now need to be scaled. Initiatives in the *Sustain* phase have scaled and drive the majority of current revenue for the organization. Initiatives in the *Retire* phase have reached their end of life and have diminishing prospects.

Using Explore, Exploit, Sustain, Retire to create a healthy product portfolio is an idea proposed by *Lean Enterprise* coauthor Barry O'Reilly. It's one of the most impactful exercises of his highly recommended Lean Enterprise workshop, and you can read more about the topic over on Barry's blog.

Explain Business Context Elements

For the majority of software engineers and many other roles, the business is a big black box. Even when the goals of the business are communicated well, an understanding of the strategic elements and fundamental mechanics of how businesses work are unclear. These employees do not, therefore, have the ability to take a true business perspective and make day-to-day decisions truly optimized for business best interests. This problem can be remedied by breaking down a business context and explaining how each piece of information fits into the overall strategic vision.

One approach to communicating a granular business context is the tech strategy pyramid (see Figure 2-1). The pyramid introduces to IT people each of the strategic decisions that lay the foundations of their technical strategy, enabling developers to take a more holistic perspective and create technical strategy fully aligned with the business vision. The layers in the pyramid are business vision, business strategy, product strategy, organization design, and technical strategy.

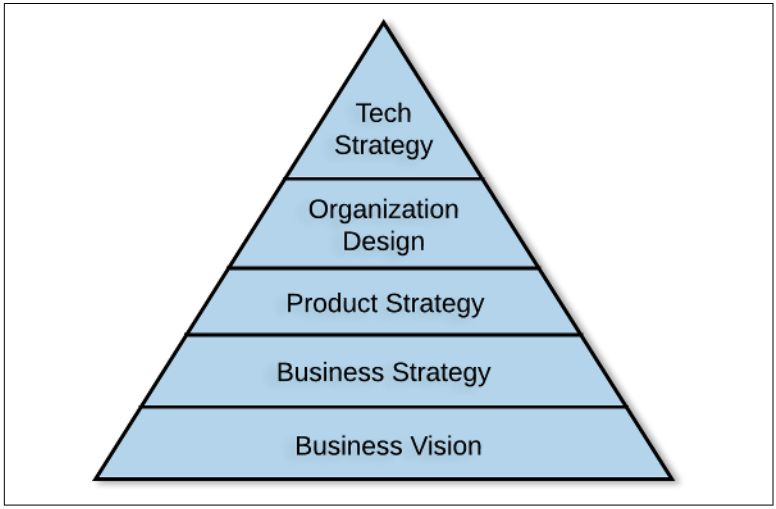

Figure 2-1. Technical strategy as a pyramid

The pyramid analogy does not imply a linear relationship. Upper layers can cause cascading changes back down to lower layers. For example, a duplication of efforts at the technology level may lead to a new team being formed to manage the duplication; a change in

organization design. A tech strategy change could also cascade down to the product strategy; a classic example that will forever be etched in the history books is Amazon Web Services. During the early 2000s, Amazon realized their IT teams were so technologically advanced with their approach to managing servers, they could sell the capability to other tech companies. AWS is now the world's dominant cloud platform, generating $12 billion in revenue in 2016. If you're not keen on the pyramid analogy, you could instead communicate your business context as a balanced scorecard.

Here's a quick summary of what each level of the pyramid is intended to represent.

Vision

To understand a business context, start with its mission and vision. A mission is the impact an organization wishes to create in the world. Everything an organization does should be driven by its mission. These are communicated via a mission statement. Here is the mission statement of the charity Save the Children:

> *To inspire breakthroughs in the way the world treats children and to achieve immediate and lasting change in their lives.*

A large enterprise may have cascading missions, where each business unit and team may have their own mission statement.

Taking the next step, an organization's vision describes where it hopes to find itself in the future. Typically, a vision looks 2–5 years ahead. The vision will be driven by the mission, but will be slightly more detailed. It provides a guiding light and an alignment point; at any time, people can ask themselves how the work they are doing contributes to the vision.

A vision is usually communicated by a vision statement. Here is the vision statement for Save the Children:

> *A world in which every child attains the right to survival, protection, development and participation.*

Notice how the vision orients the organization toward its mission. Hopefully, Save the Children will one day achieve their vision of a world in which every child attains the right to survival, protection, development, and participation. They can then create a new vision

that focuses their efforts on other approaches to improving the lives of children as set out in their mission statement.

Mission and vision should be the primary influences of an organization's more short-term cascading goals.

Business strategy

Business strategy is a high-level plan for achieving the vision, usually based around market research (e.g., analyzing large-scale trends in the market and looking for opportunities upon which to capitalize). A very public example of business strategy is Amazon's decision in 2017 to purchase the Whole Foods chain of supermarkets. This acquisition was clearly part of Amazon's strategy to become dominant in physical stores, not just virtual ones. As well as mergers and acquisitions, business strategy can involve a wide range of plays, including explorations into new markets, a focus on cost cutting, or gaining market share from leading competitors.

Situational awareness is a fundamental aspect of business strategy, argues former Canonical CEO, Simon Wardley:

> *Strategy without an understanding of the context (without situational awareness) is worthless.*

Wardley proposes his open source Wardley Maps value chain maps, which he has been developing for a decade, as a remedy for poor situational awareness. Wardley Maps are visual mapping techniques for accentuating the dynamics of an industry by identifying the key actors and influences in the value chain. Wardley argues that by understanding how all components progress from genesis to commodity, you can predict opportunities that will arise in the market. Wardley successfully used this approach to demonstrate how the cloud computing boom was entirely predictable because compute power would eventually become a commodity, as it now is.

To learn more about situational awareness and business strategy, read Simon Wardley's posts on Medium. There are also many practices for designing, exploring, and communicating business strategy, including the Business Model Canvas, SWOT analysis (strengths, weaknesses, opportunities, threats), and Porter's Value Chain analysis.

Product strategy

Augmenting the business strategy, a product strategy describes the specific product and service offerings needed to achieve the business vision. It will answer questions including: What user needs do our products not solve well? What does success look like and how can it be measured? What are the biggest unknowns and risks in the business strategy that need to be explored? As Melissa Perri, CEO of Produx Labs, asserts:

> *Most companies fall into the trap of thinking about Product Strategy as a plan to build certain features and capabilities.*

Product strategy can apply at many levels of granularity. A large organization like Salesforce (which has over 25,000 employees) may have a high-level product strategy, but each division within Salesforce (e.g., the Salesforce Marketing Cloud) may have its own product strategy. Product strategy will again be a huge influence on cascading goals but will need to be communicated in more detail using additional techniques, including the Product Strategy Canvas, the Lean Canvas, or the Value Proposition Canvas.

Creating a product strategy also involves techniques for interacting with customers to understand their needs, like user research, for example.

Organization design

Having created a product strategy, focus then switches to shaping teams in a way that will effectively execute the product strategy: organization design.

Organization design is often misconstrued as organization structure. For many people, the hierarchical org chart is the organization design. However, organization design involves the structure of the organization at different levels, the composition of the teams, even cultural elements. Accordingly, there are many elements to organization design, ranging from organization-wide to individual team level. Throughout this report, you will learn how to align your teams with your software systems—a crucial aspect of organization design, but not the only part.

One area of confusion with organization design is the interplay with product strategy. Does organization design really follow on from

product strategy and business strategy, or is an organization designed prior to strategy? Essentially, organizations are designed at a macro level based on the high-level business strategy. Each department will construct its own strategy and then design its teams. Effectively, there is a cascade. At each level of the cascade, organization design follows strategy.

Technical strategy

Technical strategy is about creating the deliverables that implement the product strategy. Technical strategy should, therefore, be driven by product strategy. Unfortunately, it's quite common to see technical strategy driven by fashionable IT trends or established best practices that are either no longer useful or unnecessary in the current context. In the worst case, some businesses do not even have an explicit technical strategy. If technical strategy is not aligned with product strategy, tremendous amounts of effort will be wasted and promising business opportunities will be missed.

Creating an effective technical strategy relies on strong communication of business goals. If software engineers understand the product strategy, and they understand the value of their work, they will feel motivated and compelled to apply their effort where there is greatest business value. If software engineers can help achieve positive business outcomes, they will be inspired to. But if they are just handed a backlog of features and expected to deliver, they will instead seek exciting technical challenges as an alternative means to gain fulfillment from their work.

External Factors

Success is not determined only by what happens inside an organization; it is dependent on many external factors. Therefore, it is important to communicate the relevance of external factors. For example, new laws, new inventions, and new social attitudes can all have a huge impact on how customers spend their money.

Market research versus design research

One important distinction to communicate is the difference between market research and design research. Understanding how these forces oppose and complement each other enables strategy to be articulated with greater precision. Market research involves iden-

tifying large scale trends in the market, whereas design research is about finding a usable solution. Put another way, market research identifies areas with opportunity for exploitation. Design research is about working with users to find solutions to exploit opportunities. When employees do not understand the importance of both activities, they can easily be convinced an idea will generate business value when only a small part of the research indicates the idea is worth pursuing.

An example of market research is identifying that $1.85 billion was spent on digital lead generation advertising in the United States in 2016, a figure that is rising year-on-year and is predicted to hit $2.12 billion in 2020.[6] These statistics scream out to marketing companies of all sizes that lead generation advertising has big revenue potential and could be a worthwhile direction to take their business. However, to exploit the lead generation market opportunities, product teams must still build products that solve genuine customer needs and provide a compelling user experience. Design research facilitates that.

By delineating between market research and design research you can avoid the common mistake of easily being swayed by one or the other. It's extremely seductive to hear you're working in an area where consumers are spending $1.85 billion dollars a year, but you still have to work hard to create a useful product. It's equally compelling to watch videos of users requesting new products or features that don't exist. Yet, if those users are in a minority, there may not be enough demand on which to build a sustainable business.

Operating context

Business context cannot be discussed in isolation. It must also be understood in terms of the operating context within which it sits. An operating context is the many external factors that can influence a business's strategy or execution, including legal, ethical, political, and even social factors, states Naomi Campbell in *The Economist Guide to Organisation Design*, "Organisation design: The alignment of all the components of an organisation in their context." For example, if new technological advancements open up new possibilities,

6 See "Digital Lead Generation Advertising Spending in the United States from 2015 to 2020 (in Billion U.S. Dollars)" for more information.

your organization should be prepared to adapt its strategy to avoid falling behind competitors. Another example would be new government legislation—this could be an opportunity to bring new products to market.

Mapping Your Business Context

Using modeling tools, you can not only communicate specific elements of your business context, but also engage all levels and skill sets in your company to collaboratively define elements of the business context. The following techniques demonstrate how you can engage all parts of your organization and create an inclusive environment that encourages innovation from the ground up.

Creating a Business Model Canvas

A Business Model Canvas teaches all employees how to focus on the most important aspects of their business, and paves the way for them to innovate. A Business Model Canvas contains nine key elements (see Figure 2-2):

Customer segments
: Who are the different types of customers the business creates value for? For example, mass market, healthcare professionals.

Value propositions
: How does the business create value for its customers? For example, fun multiplayer video games.

Channels
: How does the business communicate with potential customers to promote, sell, and deliver value propositions? For example, website, physical stores, email.

Customer relationships
: How does the business communicate with customers for after-sales issues? For example, an electronic helpdesk.

Revenue streams
: How does the business generate revenue from its value propositions? For example, transaction fees, recurring membership fees.

Key activities
: What activities are essential to making the business model work? For example, curating content.

Key resources
: What assets does the business need for the business model to work? For example, intellectual property, staff.

Key partnerships
: Which partners play a significant role in the success of the business model? For example, restaurants, technology suppliers.

Cost structure
: What are the business's major cost drivers? How are they linked to revenue? For example, salaries, procurement, licensing.

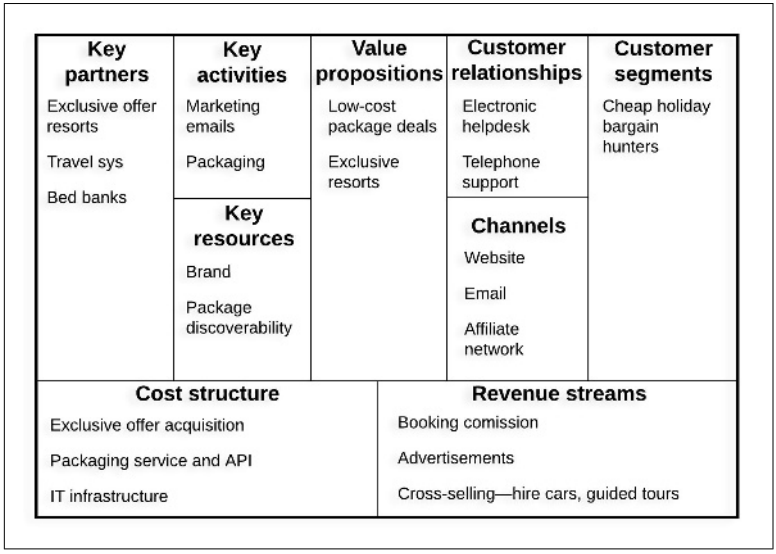

Figure 2-2. A Business Model Canvas for a low-cost travel company

The Business Model Canvas was first introduced by Alexander Osterwalder and Yves Pigneur in their best-selling book *Business Model Generation* (Wiley, 2010).

Creating a Product Strategy Canvas

Using a Product Strategy Canvas can help your teams to become more purpose-driven because the canvas encourages you to focus on the problem and the ideal target condition (the ideal target condi-

tion being a business outcome and not a list of product features). In fact, a particular emphasis of the Product Strategy Canvas is iteratively moving from your current state to the ideal target state without becoming fixed on a rigid plan. See Figure 2-3.

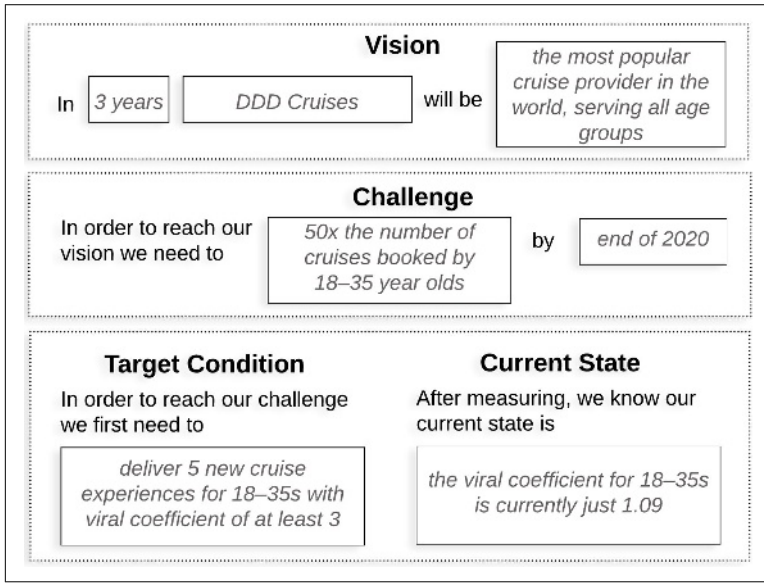

Figure 2-3. The Product Strategy Canvas of a semi-fictitious cruise holiday provider

As with other canvases and tools, the Product Strategy Canvas acts as a strong alignment point for all people in teams working toward a shared goal. To learn more, check out Melissa Perri's blog post where she introduces the Product Strategy Canvas.

Creating Impact Maps

Impact maps are a powerful tool for creating purpose-driven organizations because they force the process to begin with the problem and give everyone an opportunity to contribute to the solution.

To begin creating an impact map, start with the desired outcome. Then map out the possible actors who can help achieve the outcome. Then list the possible impact each actor could have to achieve the outcome. Finally, list the ways each actor could create their impact. Figure 2-4 shows an example impact map.

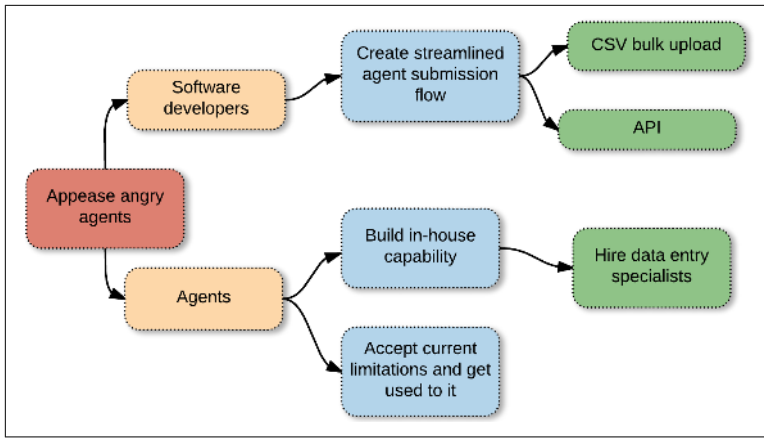

Figure 2-4. A hypothetical impact map based on the needs of a government department

Impact maps were made popular in the IT world by Gojko Adzic. To learn more about them, check out his popular book *Impact Mapping* (Provoking Thoughts, 2012).

Beyond Tools: Knowledge Dissemination Culture

Tools are unlikely to make a difference if transparency is not part of the culture. If people cannot see the strategic decisions, they cannot align their day-to-day decisions with them or suggest innovations. And if people are not given a voice in the strategic decision making, they will seek other ways to find job satisfaction. If you aspire to create a high-performance organization where teams are autonomous and software systems are aligned with business goals, a culture of transparency where information flows freely is a must.

Don't feel defeated if knowledge is not disseminated in your company. Take the initiative yourself. Regardless of your job title, start by running workshops, initially with a small group of participants you feel most comfortable with, and gradually invite larger audiences. When you do achieve successes due to greater transparency and collaboration, show off your accomplishments. Other teams will want to copy what worked for you, and the cultural improvements will spread through your organization as Javier Fernández learned

when his cheeky Business Model Canvas experiment transformed the culture in his teams.

CHAPTER 3
Analyzing Domains

> *The heart of software is its ability to solve domain-related problems for its user.*
>
> —Eric Evans

After articulating your business strategy, the journey to autonomous teams continues by learning how to identify cohesion in your problem domain—finding concepts that change together for business reasons. Aligning teams and software systems with domain cohesion minimizes organizational and technical dependencies—the holy grail of autonomy.

Learning how to analyze your domain involves looking for patterns and abstractions. Just like software can be reasoned about as patterns and abstraction so can problem domains. In this chapter, you'll see some of the most prominent patterns used for reasoning about domains, including subdomains, business processes, and user journeys. You'll start to see the trade-offs involved in deciding which of these constructs you should align your teams and software with. In the next chapter, you'll then learn practical techniques for discovering these constructs in your domain, and designing your teams and code around them.

Finding Domain Cohesion

Relationships exist between concepts in your domain. Concepts with the strongest relationships change together most often and, therefore, represent raw domain autonomy. Take the real-estate

domain for example; living room space and outdoor space have a strong relationship because they describe different physical attributes of a property. However, the concept of a purchase date relates to the transaction and not the physical property. If a new law dictated a certain unit of measurement must be used in the property description, it would affect the living room space and outdoor space, but have no impact on the purchase date.

If you align teams with organic domain autonomy boundaries, teams will naturally incur fewer dependencies and achieve greater autonomy. There are three well-established heuristics for finding organic domain autonomy boundaries: subdomains, user journeys, and business processes. None of the heuristics are perfect, but all are extremely useful.

Subdomains

Subdomains are organic domain autonomy boundaries that encapsulate one or more business capabilities. Large problem domains can be broken down into many smaller subdomains. Essentially, the strength of relationships between concepts within a subdomain is greater than relationships with concepts outside of the subdomain. Consider the domain of online music. Some businesses operating in this domain provide a comprehensive catalog of the latest and greatest music by the world's biggest and smallest artists. Their catalog is available through their own website, and also their whitelabeling API platform, allowing any business to build its own media streaming and download service.

Businesses operating in the online music domain may identify a number of subdomains, including media discovery, media delivery, and licensing. Media discovery would be the business capabilities enabling customers to discover music in the catalog. Media delivery would be the capabilities to transmit audio to customers as streams or downloads. Licensing would be the complicated business policies that enable agreements to be brokered between record labels and B2B customers. See Figure 3-1.

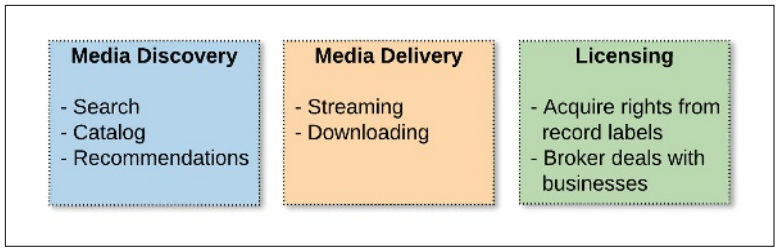

Figure 3-1. Hypothetical digital music subdomains

Subdomains accentuate business value

Subdomains should be expressed in terms of their importance to the organization. Are they core, supporting, or generic? Core subdomains represent business differentiators. More effort and innovation in core subdomains can lead to exponentially greater generation of business value compared to innovation in supporting domains where the return on investment may be small or insignificant. For example, if an online music business becomes a world leader in content discovery, end users will consistently find music that is perfect for them. Those users will make more purchases and they won't want to use competing services where they waste more of their time and find lower quality music. So more investment in optimizing search algorithms and recommendation systems could directly equate to happier customers, greater profits, and a larger market share. Contrastingly, there is less to be gained by investing in delivery. It's important that users can stream and download music, but it is unlikely to be a differentiator to competitors. You can think of supporting subdomains as akin to table stakes features.

Online music businesses may also have a customer support subdomain for managing the relationship with end users. In the domain of media streaming and downloading, customer support is also a table stakes feature—it's important, but unlikely to be a differentiator that will affect large-scale customer purchasing decisions. However, the customer support subdomain may be a generic subdomain rather than a supporting subdomain because it represents a series of behaviors and capabilities that can be reused across many diverse types of domain. Generic subdomains can often be bought (e.g., SaaS) or built in house with smaller investments than core offerings. Figure 3-2 shows how these concepts could be applied by a business in the digital music domain.

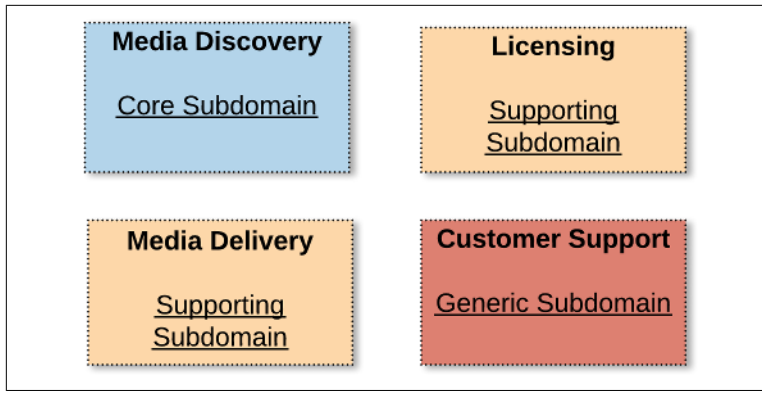

Figure 3-2. Hypothetical digital music core, supporting, and generic subdomains

Using subdomains to design autonomous teams and services

Understanding the approximate business value of each subdomain gives you powerful tools for reasoning about your organizational and technical boundaries. You should strive to avoid business bottlenecks impacting core subdomains because the more you can discover and deliver business value in core domains, the greater the overall returns for the organization. For the same reasons, you should strive to avoid introducing technical dependencies on core subdomains when architecting software systems.

When prioritizing work and deciding how to resource your teams, you should naturally gravitate toward optimizing for the continuous discovery and delivery of your core domains by allocating more people or your most capable people. At the other end of the scale, the business value of subdomains can help you to make critical build or buy decisions. If customer support and payment capabilities are not areas you need to innovate on, consider purchasing existing solutions freeing you up to focus on your core instead.

Subdomains represent cohesive boundaries in your domain encapsulating behaviors and capabilities that change together; therefore, subdomains are ideal team boundaries. If a single team owns a subdomain, changes will occur more frequently within the subdomain than with other subdomains, minimizing bottlenecks and handovers. Unfortunately, though, the size and complexity of subdomains varies so a 1:1 mapping is not always possible or even

desirable. One reason is when subdomains are too big for a team. You'll see later how partnership contexts support that use case.

Business value of subdomains can change over time

Over time, what is core to the success of an organization can change. What is core today may become commoditized tomorrow, and thus becomes a generic subdomain that can be bought off the shelf because it no longer provides a competitive advantage. For example, a startup may develop a new SaaS-based recommendations system that perfectly predicts the music any customer desires. Accordingly, all music stores and platforms should leverage the new service because it's better, more cost effective, and represents an area with no possibilities for differentiation with competitors.

In contrast, supporting domains, and even generic subdomains, can surprisingly become core. A dramatic example is Slack, the SaaS-based enterprise chat system now worth around $4 billion. Slack started life as an internal chat system used by a startup called Tiny Speck, which was building a browser-based massively multiplayer online game called Glitch. The game was a failure, but converting their internal chat system into their core offering was a massive success for the organization now known as Slack Technologies.

Not all changes to the business value of a subdomain are as dramatic as the Slack example. The moral of the story, though, is to have the mindset of constant change. Discovering subdomains is not a one-off activity that gives you perfect boundaries in your domain and identifies where you can find business value. Subdomains are a point-in-time best guess and should constantly be challenged.

User Journeys

User experience is a significant indicator of which capabilities change together. Therefore, consider aligning your teams and software services with specific user journeys, so a single team, or a small group of teams, has complete autonomy to discover and deliver the user journey. Consider eBay, for example, and marketplace business models in general, which usually have two clear user journeys: the buyer journey and the seller journey (they also have other journeys). See Figure 3-3.

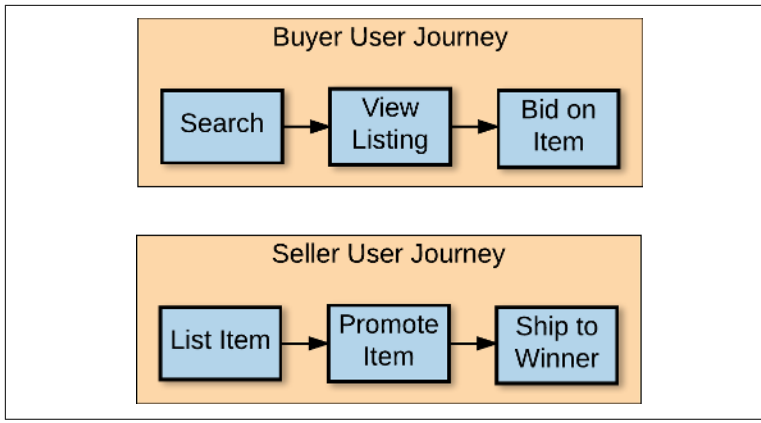

Figure 3-3. Hypothetical user journeys in an eBay-like auction site

A user journey is a sequence of steps taken by a user in order to complete an action or fulfill a goal. User journeys can begin and end with activities in the real world, not only involving software systems. Nowadays we live in a service design world where single user journeys span many devices. Understanding the flow across all interfaces is paramount to building successful products with a great user experience. The design of teams and software architecture has to be primarily driven by enabling these superior full-service experiences. The ability to constantly improve experiences by practicing continuous discovery and delivery is, therefore, a necessity.

Many user journeys will be too large for the ownership of a single engineering team. Consequently, user experience consistency and team autonomy are both at risk. A response to this quandry is the Centralized Partnership pattern (illustrated in Figure 3-4) presented by Peter Merholz and Kristin Skinner in their book *Org Design for Design Orgs* (O'Reilly, 2016). The Centralized Partnership specifies that a single design team owns each user journey, supported by multiple engineering teams. The goal is to enable highly consistent UX and highly autonomous teams.

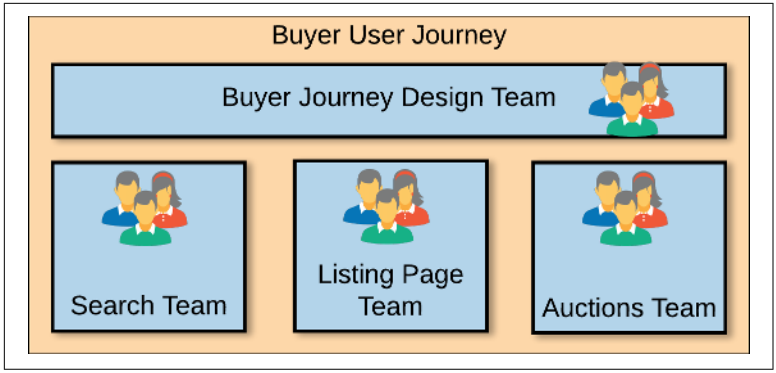

Figure 3-4. The Centralized Partnership (source: Merholz and Skinner)

User journeys can be device-specific instead of spanning many devices, and this can have a big impact on the design of your teams and services—sometimes it may make sense to have a dedicated team per device whereas other times it may be more effective to have teams who own specific capabilities used by multiple devices.

One user journey nuance to keep in mind is that not all users are human. Use cases involving technical integrations with external systems and actors can also be considered user journeys that teams could be aligned with.

NOTE

You will see technical patterns for splitting a single user journey across multiple teams in Chapter 6. You will also see patterns for creating dedicated experience teams who own the UX for a single device.

Business Processes

Understanding key business processes is the secret to creating autonomous teams and services in some domains because the business process is the biggest influence on what changes together. For example, sometimes the high-level business process is largely stable, yet the rate of change within individual business process steps can be high. Accordingly, aligning teams with individual process steps may lead to the highest levels of autonomy and innovation.

One example of aligning by business processes is the government agency mentioned in Chapter 1. Their high-level business process was Review, Resubmit, and Renegotiate. The process was driven by

government legislation, which changes infrequently, on the scale of years or decades. However, each process step (Review, Resubmit, Renegotiate) was continuously being iterated to provide a more effective service to citizens and a better user experience—a strong indication that carving out team and software boundaries aligned with business process steps could lead to the highest autonomy on core offerings.

Comparing Subdomains, User Journeys, and Business Processes

Subdomains, user journeys, and business processes are all invaluable heuristics in finding the domain cohesion needed, but their differences can sometimes be subtle. In simplified terms, a user journey can span multiple subdomains and vice versa. A business process can also span multiple subdomains and user journeys. In fact, a single business process step can span multiple subdomains. Sound confusing? Hopefully Figure 3-5 helps to clear up the confusion.

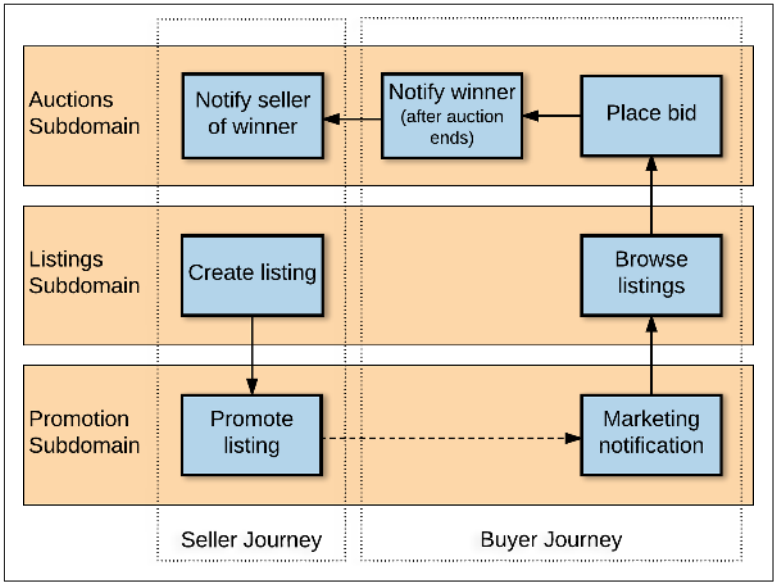

Figure 3-5. Comparison of subdomain, user journey, and business process in online auction domain

Deciding whether to align teams and technical components with subdomains, user journeys, or business processes is a challenge.

There is no best practice. The remainder of this report will show you techniques for identifying the best choices in your domain based on your business context.

Solution Space Building Blocks

Subdomains, user journeys, and business processes are powerful constructs for discovering autonomy in your domain. Solution space building blocks enable you to carry the autonomy they represent into your teams and software architecture, by aligning them with the domain and minimizing accidental complexity.

Bounded Contexts

Bounded contexts are boundaries in your software architecture. The goal is to align bounded contexts with subdomain, business process, or user journey boundaries so cohesion in your domain is reflected as cohesion in your software. Ergo, things that change together in your domain are modeled together and isolated in your software.

Bounded contexts are necessary because it's not always possible to directly model subdomains or processes in code due to the messy reality of complex human systems and complex software systems. For example, media delivery may be too large an initiative for a single team to own, so the subdomain would be represented as two bounded contexts in code: the streaming context and the downloading context (see Figure 3-6).

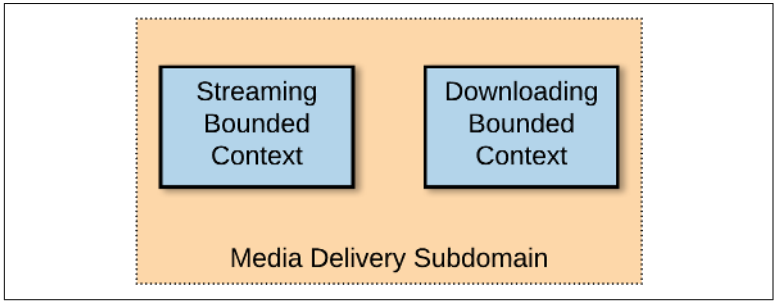

Figure 3-6. A subdomain composed of multiple bounded contexts

NOTE

Arguably, if streaming and downloading each have unique concepts and phrases associated with them, they are linguistic boundaries and should, therefore, be separate bounded contexts anyway, regardless of belonging to the same subdomain or not.

The relationships between bounded context and user journeys, and bounded context and business processes, also lack precise definition as shown in Figures 3-7 and 3-8. Domain and business context should drive the design.

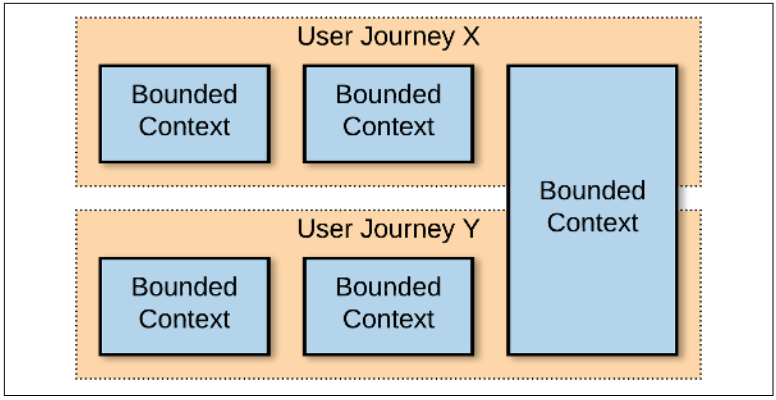

Figure 3-7. User journeys versus bounded contexts

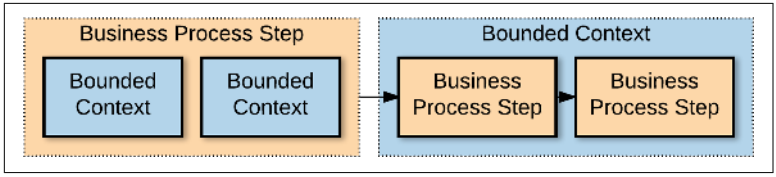

Figure 3-8. Business process steps versus bounded contexts

When done well, bounded contexts make domain concepts explicit in architecture and code, minimizing the cost of translation between the business and technical worlds. Bounded contexts and the code contained in them should be named after the business concepts they represent. By minimizing the costs of translation, bounded contexts accelerate product development in multiple ways: the teams that own bounded contexts have more autonomy because they own autonomous business concepts, and use of domain language in code and conversation results in less ambiguity and fewer misinterpretations of requirements.

Autonomous Contexts

One the biggest limitations of bounded contexts is that they have gained a widespread reputation as merely a naive means to align code with the domain. Consequently, team autonomy is overlooked. However, team boundaries and context boundaries are inextricable—they should be co-designed to maximize a team's autonomy to continuously discover and deliver.

To encourage team autonomy as the number one goal when designing organizations and software architecture, use autonomous contexts as the most granular level of abstraction in your design. Autonomous contexts are boundaries representing specific product or business capabilities and everything needed to continuously iterate them, including a single autonomous team with all of the necessary skills, and all of the required technical pieces (e.g., websites, APIs, and databases). See Figure 3-9.

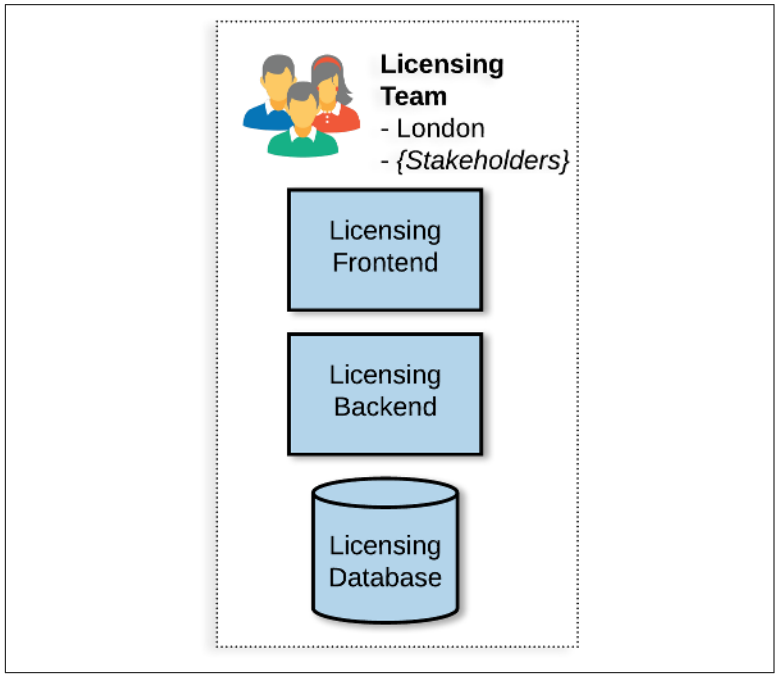

Figure 3-9. An autonomous context

When you explain your design in terms of autonomous contexts, you are stating that autonomous teams are your primary focus, and

that you are conscious of the interdependence of business, domain, organizational, and technical factors affecting your design.

What Are Cross-Functional Teams?

The definition of a cross-functional team used by this report is a team comprised of all the skills needed to achieve its business outcomes including product managers, software developers, infrastructure experts, desigers, user researchers, and business stakeholders.

Partnership Contexts

It's not always possible for teams to be fully autonomous. Invariably, they must collaborate to deliver higher-level business capabilities. Partnership contexts are a logical construct giving you the power to design, make explicit, and articulate the collaboration between autonomous contexts in your organization.

Partnership contexts help you to accentuate essential communication pathways that should exist between teams. When teams understand they are part of a partnership context, their default mindset will be to work closely with other teams in the partnership. Equally, teams will treat excessive collaboration or technical dependencies with teams outside the partnership with caution, questioning whether it is an essential dependency or accidental complexity that can be removed to improve autonomy. See Figure 3-10.

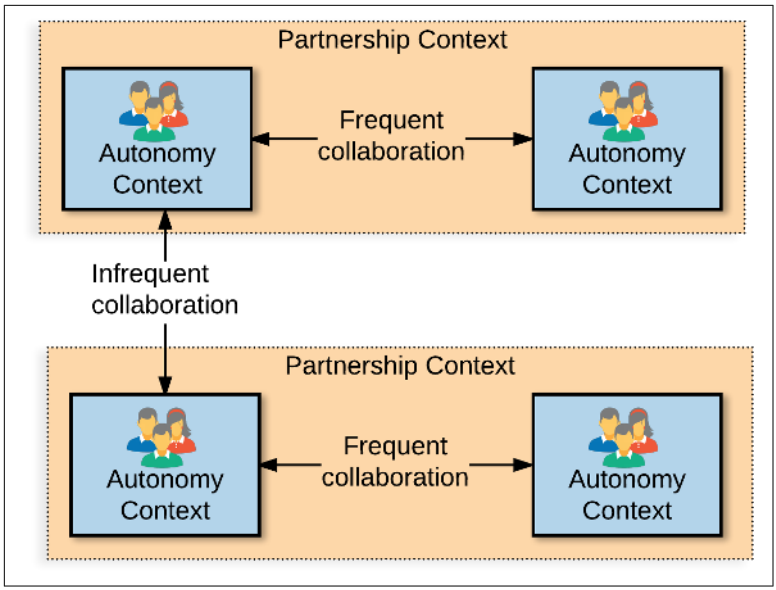

Figure 3-10. Partnership contexts

In Spotify nomenclature, partnership contexts are analogous with tribes, groups of teams (aka squads) who work closely in a related business area.

One approach to identifying partnership contexts in your organization is to align them with large subdomains composed of multiple bounded contexts. Each bounded context becomes an autonomous context owned by a single team, and the subdomain boundary is the partnership context boundary. Another indicator of the need for partnership contexts is when a single backlog grows too large for a single team. Rather than having a large, ineffective team, use the techniques presented in the next chapter to identify individual contexts that can then form a partnership context.

Don't stop at partnership contexts. Organizations can be reasoned about at multiple levels of abstraction—for example, business units, organizations, departments, and clouds (see Figure 3-11). There are many such terms to describe higher-order groupings of teams, although each phrase can have drastically different semantics in different organizations.

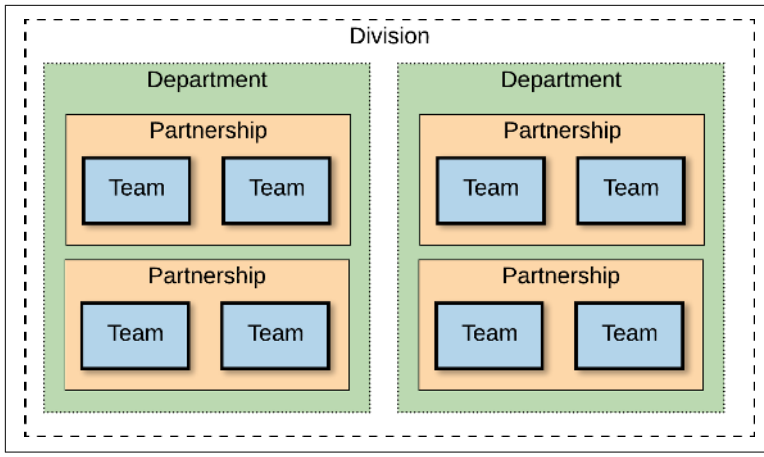

Figure 3-11. Organizations have many levels of cohesion

Finding the right boundaries can seem overwhelming. It's more art than science. If you are ever unsure, remember the golden rule of autonomy: the rate of change inside boundaries should be greater than the rate of change across boundaries.

CHAPTER 4
Discovering Contexts

> *Finding service boundaries is really damn hard…there is no flowchart.*
>
> —Udi Dahan, *Service-Oriented Architecture Expert*

In the previous chapter you learned abstract patterns for modeling autonomy in the problem space. In this chapter, you'll learn collaborative techniques for discovering your problem domain. Critically, you'll also learn to develop a modeler's most important characteristic: never being satisfied with the model.

Multiple perspectives of the problem space are needed to create the best solution. In this chapter, you'll see outside-in approaches, starting with the goal and working in toward the domain. And you'll also see inside-out approaches, starting with the domain and working toward the goal. Expert modelers combine both approaches with knowledge of the business context (presented in Chapter 2).

Crucially, representatives from all sides should be involved in both perspectives. Typically, you'll find organizations will split the responsibilities; managers and coaches will take the systems approach and focus more on teams, while software engineers focus on the architecture of the software. But creating high autonomy requires high alignment between the organization design and the software architecture. You cannot achieve autonomy if managers exclusively design teams and engineers exclusively architect software systems.

If high performance is your goal, you need to accept that the design of teams and the design of software systems are inextricable. You must design cross-functionally, collaboratively, and conscientiously.

Explore Core Use Cases

To uncover the subdomains, user journeys, and business processes in your domain, start by exploring your core use cases. Review your strategy communication tools—for example a value proposition canvas—and identify the use cases that your customers care about the most.

The following examples of exploring use cases will be based on an online music platform. They key details of the business model are as follows:

Customer segment	Value proposition
Individual music consumer	Discover, download, and stream all of the best music in the world.
Music platform builder	Build a media streaming and downloading service on our fully managed platform containing access to all of the best music in the world.

Key partner	Reason
World's biggest record labels	They provide all of the best music that end users want to hear.

Key activity	Reason
Making it easy to find the best music	Customers will otherwise use competing services where they can find music they like easier and faster.
Enforcing licensing restrictions (e.g., regional)	Record labels will pull their content from the platform if it is being accessed illegally (e.g., someone in the UK listening to a track that has not yet been released in the UK).

Based on the business model just outlined, and further sources of information including detailed discussions with domain experts and business stakeholders, a cross-functional team collectively agreed on the following top five use cases:

- Individual consumer: Finding and purchasing a song
- Individual consumer: Streaming a song
- Individual consumer: Downloading a song

- Music platform builder: End consumer finding and purchasing a song
- Record label: Ingesting new music into the platform with licensing restrictions

Outside-In Use Case Modeling

To begin modeling use cases, it's usually best to start from the outside. Take the perspective of a user and understand how the system would ideally behave from their perspective. Taking this approach will help you to empathize with customers, understanding the value each use case provides to them, and clarifying how the software system you need to build should behave in order to deliver value.

Jobs to be Done

The Jobs-to-be-Done framework (JTBD) is a tool for exploring use cases from the customer perspective in order to identify potential customer value. JTBD begins by identifying a job, task, or goal your customer wants to achieve and why they want to achieve it. Therefore, JTBD is an ideal starting point when modeling use cases because it starts with value and works backward. JTBD can be expressed as job stories. Job stories have the format *situation*, *motivation*, *expected outcome* using the following template:

> When <*something happens*> I want to <*do something*> so I can <*achieve some goal*>.

The following are job stories created by the team derived from user research:

Finding and purchasing a song
: When an individual music consumer is searching for new music, they want to quickly find something relevant to their tastes, so they can continue their original task with the pleasure of new music, without wasting lots of their time finding it.

Downloading songs
: When an individual music consumer has purchased music, they want to download it so they can play it via their favorite devices

and media players, so they have the flexibility to enjoy the music however and wherever they like, such as the gym.

As you can see in these examples, job stories can contain a lot of context, and that's exactly their purpose; job stories try to capture as much context and motivation as possible to create a clear shared understanding of the problem, rather than jumping into a solution.

If your team is practicing continuous discovery, many of the insights you gain from user research sessions will feed into your JTBD. Not only will that allow you to more accurately model use cases and work back from customer needs to system behaviors, it will give your entire team empathy with customers, encouraging whole-team innovation.

JTBD goes far deeper than the example presented here. To find out more and get started with JTBD, check out *jobstobedone.org*.

BDD

Behavior-driven development (BDD) is a set of practices for describing how users interact with the products and systems you build. BDD naturally synergizes with tools like JTBD; JTBD helps you frame the problem and the user need, while BDD helps you translate those needs into the ideal interactions between users and software systems in order to satisfy the need.

The "behavior" in BDD refers to the behavior of the system you are building—encouraging you to focus on how your system behaves from the perspective of a customer, and how those behaviors satisfy user needs, creating customer and business value.

BDD is essentially a collaboration tool. The goal of BDD is rich discussions that lead to clear insight and shared understanding, giving teams a rich vision of what they need to build. However, BDD does have iconic artifacts: BDD scenarios. BDD scenarios are structured, plain-English steps that codify the required behaviors of your system, expressed from the perspective of your user. For each of your JTBD, you would, therefore, have one or more BDD scenarios describing the variations in each use case—the happy paths, the error paths, and so on.

Example 4-1. Hypothetical job stories for a business in the online music domain

Job: Finding and purchasing a song

Scenario: Individual consumer: Buying a song available in the catalog
Given I am in the UK
And I search for "Madonna - Like a Virgin"
And I add the song "Madonna - Like a Virgin" to my basket
When I check out using my real address and a valid credit card
Then I receive a purchase confirmation
And the song "Madonna - Like a Virgin" is in my locker

Scenario: Record label: Consumers cannot buy songs unlicensed in their region
Given the song "Humpty Dumpty - All The King's Horses" is not available in the UK
When a consumer in the UK searches for "Humpty Dumpty - All The King's Horses"
Then "Humpty Dumpty - All The King's Horses" should not be in the results

There is much debate about whether BDD scenarios should be translated into the automated tests written in code. That's a decision for your teams to make. The purpose and benefits of BDD for modeling use cases are unquestionable though; focus on the behaviors of your system and how they satisfy user needs before you dive into the code.

Inside-Out Use Case Modeling

Inside-out modeling techniques encourage you to build a rich understanding of your problem space. The goal of inside-out modeling techniques is to design the inner workings of your system required to deliver the external behaviors you have identified with outside-in modeling techniques. Inside-out modeling is also valuable for clarifying intricate domain policies that may already exist in the real world and need to be modeled in the software system.

In a greenfield scenario, it's likely you will want to start with outside-in modeling (i.e., you'll want to focus on the user need and the business value and the behaviors your system must provide to achieve them). In a brownfield scenario, you may decide that inside-out

modeling (i.e., trying to understand what the system is actually doing) is more important. However, there are no rules. As long as you come at the problem from both angles, you'll gather the vital information necessary to design autonomous organizational and technical boundaries.

Domain storytelling

If you're unsure how to model the internal behaviors of your system, start with domain storytelling. Domain storytelling is a visual notation with four basic building blocks, allowing you to model the processes in your domain and software system representing your use cases. By visualizing use cases as domain stories, domain cohesion is accentuated—it becomes clearer to see which concepts are related together and involved in the same use cases. This information is gold dust for identifying boundaries in your system. You can even represent your context boundaries on domain story diagrams.

The four building blocks of domain stories are actors, work objects, numbered activities, and annotations. Actors are people or systems who communicate using work objects as part of activities. Activities are numbered so you can understand the flow of the use case. Finally, annotations are used to provide additional detail that is important to understand but not easy to express as part of the story. Figure 4-1 shows the story of a consumer searching for a licensed track in the online music domain.

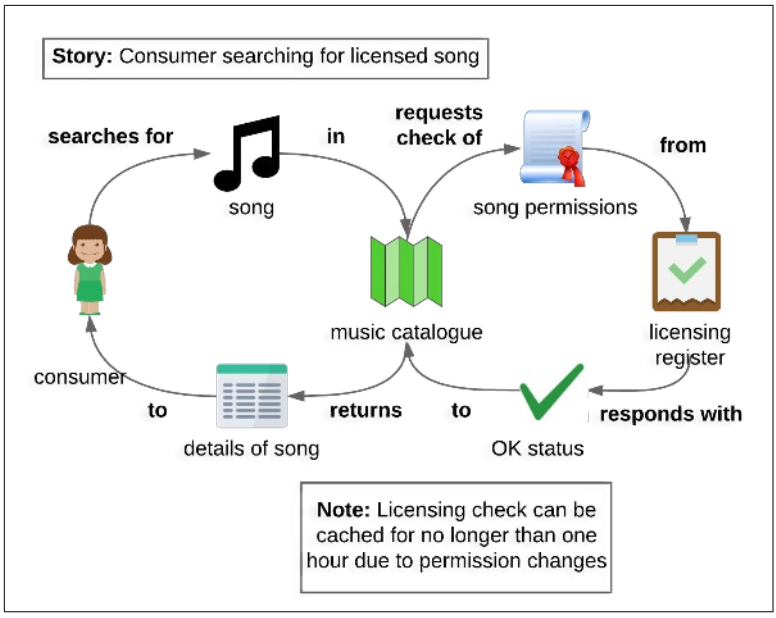

Figure 4-1. An example domain story for the online music domain

It's possible to create domain stories at various levels of detail. You may be able to express entire use cases as a single domain story, or you may require multiple domain stories to express a single use case in enough detail from different angles. A good default to start with is a single domain story per BDD scenario.

To represent contexts on your domain stories, draw boxes with dashed lines around the behaviors that belong to individual contexts (Figure 4-2). If you haven't yet started to design the contexts in your system, gathering together all of your domain stories and sketching hypothetical boundaries is a clever place to start.

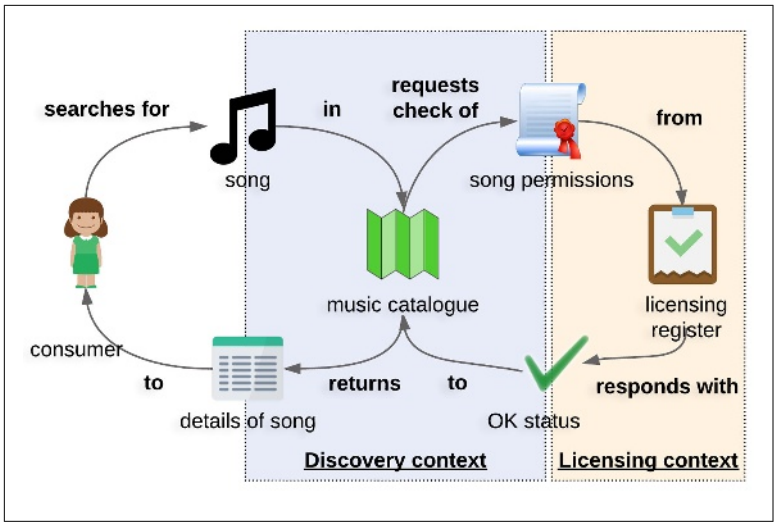

Figure 4-2. Showing context boundaries on domain stories

You do not need specialist or commercial software to create domain stories. You can use any general-purpose drawing software, or even an old-fashioned whiteboard and sticky notes. To learn more about domain storytelling, visit *domainstorytelling.org* and join the domain storytelling community.

Domain use case diagrams

For longer or more complex use cases, domain stories may be too granular to visualize the entire use case clearly. Instead, zoom out a level and create informal domain use case diagrams showing the interaction between contexts in your system.

Similar to domain stories, domain use case diagrams show the flow of activities for a given business process, use case, or scenario. They can be used to visualize existing processes or suggest potential new ones. However, domain use cases operate at the level of bounded contexts using the same familiar notation: actors, external systems, activities, and contexts. Figure 4-3 shows a domain use case diagram for the online music domain.

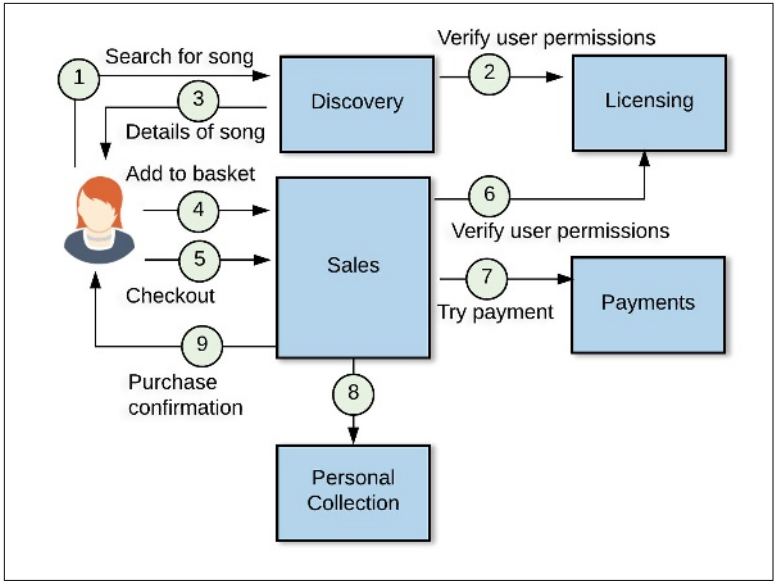

Figure 4-3. A domain use case diagram

Notice the level of detail in comparison to a domain story. A domain use case diagram is at a higher level. Each step on the domain use case diagram could make up a single domain story composed of multiple smaller level steps. Clarity is the arbiter. Understand which level of detail you want to communicate in a diagram and keep the noise down. If you have more than 10 steps on a diagram, it's probably trying to communicate too much. Consider creating higher-level diagrams or chopping up diagrams and adding links between them.

Other techniques

Event storming, which we'll discuss in Chapter 5, is a powerful techniques for visually exploring entire domains. You may also wish to look at formal techniques, including UML Sequence Diagrams and UML Use Case diagrams, or ad-hoc diagrams. Finally, don't forget the old-fashioned approach: gather a mix of people around a whiteboard for a game of pretentious arm waving and drawing pretty colors.

Create Multiple Models

> *All models are wrong; some are useful.*
>
> —George Box

When you understand your business context and you have started exploring your domain use cases, you are ready to start designing your organizational and technical boundaries. You should adopt an experimental mindset, considering multiple different models and not being satisfied with the first idea. In fact, you should never be satisfied with your boundaries. As your operating context changes or you gain new insights from the domain or your technical implementation, you should be prepared to realign boundaries in order to improve how you create and deliver business value.

Hypothesize with Context Maps

Context maps are used to visualize the contexts in your system. They are a common tool not just for visualizing existing context boundaries, but also for hypothesizing about new ones. Whereas use case diagrams focus on how contexts interact to carry out individual use cases, context maps provide a single view of all of the contexts in your system—your organizational and technical boundaries, combined with valuable contextual information, including the relationship between teams, how frequently they communicate, and the general business value of each context.

The context map in Figure 4-4 was designed to accentuate the areas of ownership each team has, and the relationships between teams.

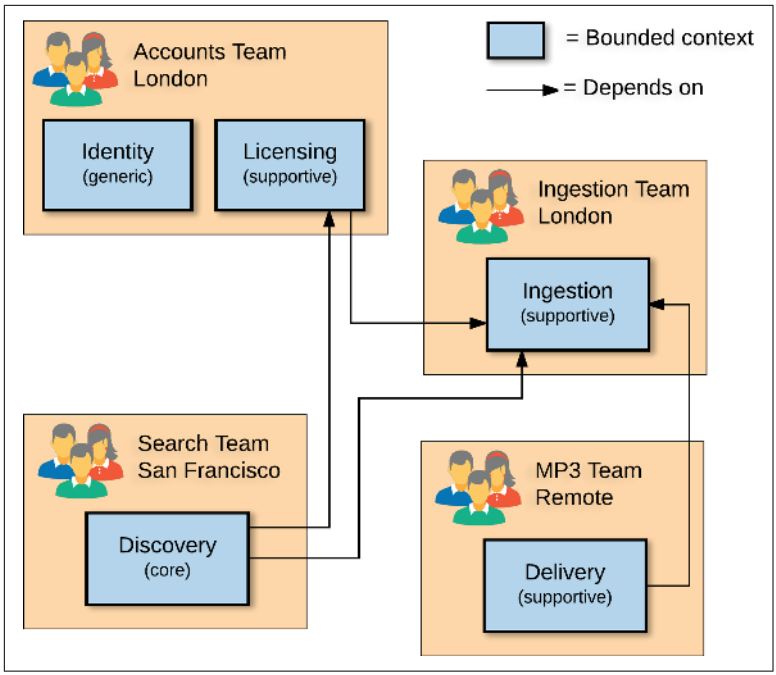

Figure 4-4. A hypothetical context map for a business in the online music domain

There is a wealth of information that can also be shown on context maps, including:

- Type of relationship (Does one team dictate to the others, or are they partners?)
- Reasons for integration (What do the lines between contexts signify?)
- Subdomains (How do contexts align with subdomains?)
- Level of debt (Are contexts legacy, in other words, do they have high technical debt and other things you'll need to consider?)
- Communication bandwidth (How frequently do teams communicate?)
- Technical integration strategies (RPC, pub/sub, etc.)
- Multiple teams owning a context or multiple services within a context

Context maps are an extremely flexible tool; in fact, they are more of a guideline than a standard. But they do focus you on the right questions and conversations. Visualize the most important aspects that affect the design of your teams and software systems. If you're unsure, go a bit too crazy and retrospectively chop out information that you decide isn't so important. Ideally, gather a range of technical and domain experts around a large whiteboard in close proximity of some tasty snacks and fine coffee. And remember to create multiple models, because all models are wrong.

Apply Domain Heuristics

As you start to design and refactor your boundaries, there are a number of heuristics that can help you improve your models and choose the most useful. You can use these heuristics to initially identify boundaries in your domain stories and use case diagrams, and also to continually review your boundaries in search of better ones.

Language

Language is traditionally one of the most powerful heuristics used by domain-driven design (DDD) practitioners for identifying cohesion in problem domains. Focusing on language is one of the best ways to find boundaries in your teams and code. Look for words and phrases that are used together, and look for different meanings of the same word in different parts of your domain. Then encapsulate those contextual semantics with clear boundaries of delineation and ownership.

Consider the concept of a song in the online music domain. In different contexts, the word "song" means different things. In the sales context, it is some kind of thing that can be purchased and has a price. Zoom over to the delivery context and "song" has drastically different semantics. In the delivery context a song is something that can be transmitted over the internet to users in a variety of formats, including MP3. Now think about the discovery context. The semantics of a song in the discovery context would involve something that has information/metadata associated with it (the artist, a release date, a record label, etc.).

In Figure 4-5, the different definitions are related; they refer to the same thing in the real world: a song. They refer to different views of the song or different phases of its lifecycle. However, in some

domains, the same word can have different meanings because it refers to unrelated concepts. For example, the word "artist" may also be used in a different part of the system to describe the person who created the artwork for a song rather than the musician.

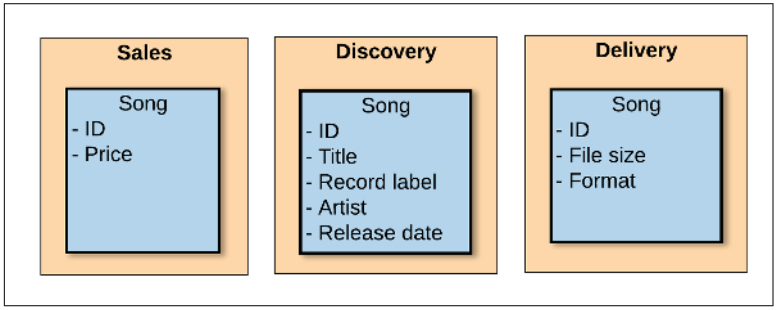

Figure 4-5. Contexts as linguistic boundaries

Used carelessly, language can lead to great ambiguity, confusion, and unnecessarily wasted effort. Used studiously and diligently, language can be the rock solid foundation for creating high autonomy in your organization. Treat your context boundaries as linguistic boundaries; because words have precise meaning within a context, strive to clarify that context whenever you use a phrase in code, conversations, or diagrams. Encourage everyone else to be passionate about linguistic precision, too, whether you are a domain expert or software developer.

Domain expert boundaries

In large, complex problem domains, you will often find multiple domain experts with different areas of expertise involved. Naturally, you are likely to find that domain experts with different areas of expertise have an interest in different parts of the domain. By looking at the language used by each domain expert, and the parts of the domain they care about, you can get a strong idea of where cohesion hides. These insights are strong indicators of autonomy.

Consider the curation subdomain in the online music domain. Domain experts there are knowledgeable about music—they create playlists for music lovers. Conversely, in the licensing subdomain, a domain expert would have specialist legal knowledge and be well trained in managing relationships with record labels.

Business process steps

In the previous chapter, you saw the benefits of aligning teams and software architecture with business process steps in domains where the rate of change within each business process step was high, and the rate of change between business process steps was relatively low. How do you know if business process steps are the best boundaries in your domain, and how can you avoid making the common mistake of identifying a business process step that isn't actually a business process step?

If you are creating domain user stories and use case diagrams, or similar visualizations, you should naturally uncover business processes. Ensure that you follow up the visualizations with an inquisitive mindset; continue to ask domain experts to name things. If you see a sequence of events that results in a specific business outcome, ask the domain expert to name it. It's not uncommon to uncover hidden business processes and policies that don't even have a name.

In Chapter 5, we'll introduce event storming, arguably the best approach for identifying implicit and explicit business processes by mapping out systems and processes as a series of events.

The following scenarios are guidelines showing how to define and organize for different types of business process.

Short business process, core to domain. If your business process has a small number of events (say, around five) and is a key process for the business (i.e., generates a large amount of revenue or delivers a value proposition), you may have naturally found the ideal boundaries for your contexts: each event. There are three factors to look for to gain confidence:

- Is the overall business process highly stable—in other words, does the sequence of events change infrequently?
- Is the interface between each event relatively stable (the data shared)?
- Can the responsibility of each event be managed by a single team?

If you can answer "yes" to all of these questions, it's a good sign that each event is a separate step that could be owned by a single team. You may even find that multiple steps can be owned by a single team. However, pay special attention to the final point; if a single step is so large that it requires a single team to own it, it may be more than just a business process step.

Short business process with low granularity. If your business process has few, but coarse-grained steps, it may be a sign you need to decompose each step into the individual activities. You may discover there are implicit concepts hiding, and that business process steps aren't really business process steps. This can often occur when people use specific domain terms loosely. For example, people may refer to the "Ingestion" business process step, which leads system designers to create an ingestion team and context. However, more detailed analysis would uncover that ingestion itself is made up of a number of smaller processes, and it may make sense for steps in those processes to form their own contexts for a variety of business, technical, or domain reasons.

When you are unsure how granular your model of business steps should be, work back from value propositions and think about team size. If the entire step is manageable by a single team, let team size dictate the boundary. However, if certain substeps are more key to delivering value propositions and would benefit from the dedicated attention of a single team, isolate valuable substeps into their own contexts accordingly.

Long business process with many events. If you find your business process has many steps (e.g., more than 20), you need to find which are the most cohesive in order to find a higher level of abstraction. Essentially, you may have implicit concepts in your domain. Talk to domain experts and business experts to try and find a name for specific groups of events. You may want to ask them, "Is there a name for this subsequence of events?" or something similar. You may even need to work with them to create a name.

An example of finding a higher level of extraction would be the reverse of the previous ingestion example. You may begin with a number of granular steps, later to discover the higher level business process step of ingestion.

Data flow and ownership

The way in which data flows in your domain can be a big clue about boundaries. Data is, essentially, a coupling. Where two contexts rely on the same piece of information, which must flow from one context to the other, the dependency requires constant management. The originating service may want to amend the data format, stop supplying the data altogether, or introduce other types of breaking change. Even worse, there may be multiple consumers of the data. So clearly, when discovering cohesion in your domain, when searching for things that change together, you have to pay special attention to data.

In the online music domain, a discovery context could be composed of search and catalog. Both rely on the same data. The catalog needs to make songs browsable, and the search service needs to make all of the fields on a song searchable. Therefore, those individual capabilities are highly cohesive (i.e., they change together) due to data, so they have been modeled cohesively as a single context. That doesn't mean search and catalog should always be modeled as a single context in every domain, or even in this domain as the business context changes and the organization scales.

Sharing data does not necessitate two services being reliant on the same physical database. The implication is that whenever the data representation of a concept changes, all consumers of that data must accommodate the change, increasing coordination costs and reducing team autonomy.

Consider the government agency introduced in Chapter 1. Whenever a change was made to the Resubmit or Renegotiate context, a corresponding change had to be made in the Case Management context, because whenever Resubmit or Renegotiate collected new information from citizens the exact same information had to be presented in the same way to the case worker using the Case Management system. In this example, the data dependency is hinting that the design is wrong; by co-locating all the data that changes together, the dependency can be removed. For this scenario, that means decomposing the Case Management context into a composite application, and letting the Resubmit and Renegotiate contexts have full encapsulation of their data.

Understanding the flow of data is not something you will be able to design up front. All of the patterns and principles in this chapter rely on continuous design, especially data. Until you start building the system and bringing it to life, you can never be certain of the precise data formats and how they will change over time.

CHAPTER 5
Designing Antifragile Systems

> *Antifragility is beyond resilience or robustness. The resilient resists shocks and stays the same; the antifragile gets better.*
>
> —Nassim Nicholas Taleb

Thinking in systems is vital in a world where services need to be designed seamlessly across many devices, and where organizations are becoming increasingly networked. Reasoning about all of the interacting elements in systems of people, systems of software, and systems of services, and making choices that optimize for system-level goals is undeniably a necessity for individuals working in high-performing organizations. Systems must continually sense and respond to changing market conditions and new opportunities—they must be inherently antifragile.

In this chapter, you'll see how leading enterprises take a systems perspective to optimizing the alignment of organizational and technical boundaries. You'll see how to apply concepts like the Theory of Constraints to enhance your models and determine the most useful. You'll see the synergies between the domain-driven design approaches presented in Chapter 4 and the systems approaches presented here. Conclusively, you'll see how both perspectives are vital in a holistic design process optimized for continuous adaptation. Then in Chapter 6, you'll see advanced technical patterns for building antifragility into your software architecture, enabling full slices of autonomy right through your business.

Coevolving Organizational and Technical Boundaries

> *Organizational architecture is as important as software architecture. They are symbiotic.*
>
> —Alan Kelly

When Fortune 500 CEOs cite "organization [is] designed inappropriately for digital" as one of their most significant challenges,[1] they are expressing fear of their fragile organizations. They fear fast-moving, antifragile competitors who constantly exploit emerging opportunities.

In order to create antifragile organizations, the relationship between organizational boundaries, technical boundaries, and business context is one of the most important factors to comprehend. Changes to team boundaries can have drastic implications on your ability to create software effectively. Changes to software architecture can speed up slow design progress of the entire business. Further, boundaries in both the organizational and technical landscapes can quickly become harmful if the operating context changes. Great attention must be paid to the intricate relationship between operating context and organization design, as illustrated in Figure 5-1.

1 See "Cracking the Digital Code: McKinsey Global Survey Results" for more information.

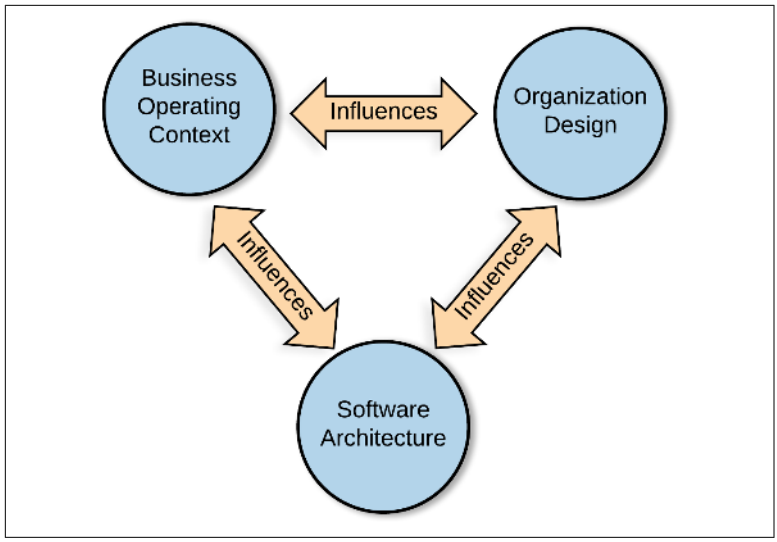

Figure 5-1. The symbiotic relationship between business operating context, organization design, and software architecture

Technical Realities Influencing Operating Context in Online Travel and Digital Marketing

In summer 2013, the operating context of a UK-based online travel agency changed drastically; they anxiously observed competitors critically gaining market share due to more powerful recommendations systems. Fighting to avoid collapse, improved vacation discoverability became the apprehensive agency's top priority. Consequently, their technical architecture had to quickly evolve.

The travel agency needed to improve their high cost and slow rate of change. Their monolithic system was deployed once per month, involving one whole week of manual regression testing. With their business at risk, they urgently needed to rapidly iterate on new recommendations capabilities. Attempting to stabilize the monolith in order to enable faster product development cycles was futile—it would require years of people hours to pay back the inordinate level of technical debt that had accrued.

Strategically, the agency focused on splitting out and cleaning up just their recommendations capabilities. They broke out a new autonomous context (Figure 5-2), owned by an autonomous team in under three months. An additional two months later, they were

delivering new recommendations enhancements on a daily basis fueled by continuous discovery.

By focusing resources on aligning organizational and technical boundaries with core business differentiators (recommendations) following alarming changes to their operating context, the travel agency slowly clawed back market share. Had they not focused on creating autonomy where it mattered most, the agency's fragility would have led to shipwreck.

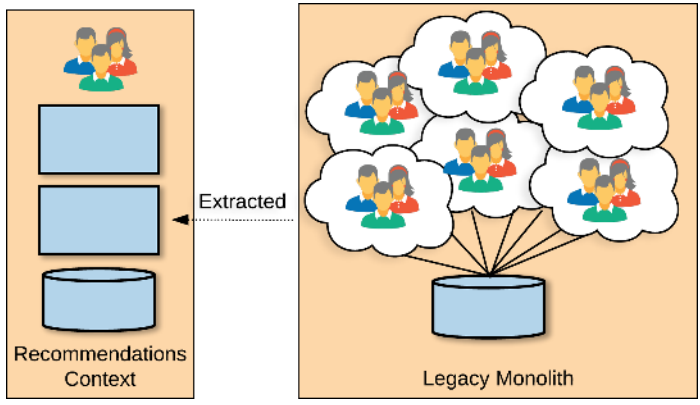

Figure 5-2. Extracting highest value capabilities from the monolith

In contrast to the travel agency, a digital marketing outfit, leaders in their niche, were forced to lower their ambitions due to changes in their technical landscape, despite no external changes to their operating context.

In late 2015, two years after being acquired by a big-name enterprise, Product & Engineering at the marketing startup called a crisis meeting. Every piece of work was now taking unbearably long to implement, causing many sleepless nights for both engineers and managers. The tipping point had finally been reached: the engineers nervously confessed that the system had been hacked together in the startup years. Unfortunately, they confessed, there were *unimaginable levels of technical debt*.

After many frank and uncomfortable meetings, the business had to put its grand ambitions on hold and readjust its vision, descoping 80% of the product backlog, reorganizing teams around delivering the highest value priorities, and paying back the most damaging technical debt.

Maximizing Knowledge Sharing Between Teams

When boundaries need to evolve for business prosperity, it takes insights in one part of the organization to trigger changes in another. Those insights can only arise when people have situational awareness of the wider organizational context: aligned autonomy. Therefore, it's vital to build a culture that encourages the flow of knowledge between teams and across the organization.

Improving visibility across teams, especially as organizations scale, is notoriously challenging. The problem was acknowledged by 43% of respondents in the 2017 State of Agile Report who cited "improved project visibility" as one of their main reasons for adopting agile. At the same time, 69% stated their reason for adopting agile was needing to develop products faster. Developing products faster with poor visibility of projects across teams leads to fragile organizations and inconsistent user experiences, as each team naively pursues their own best interests as fast as they can.

> **NOTE**
>
> In Chapter 2, the focus was on communicating information down the organization, from management to delivery teams. The focus here is spreading knowledge broadly across autonomous teams so they can work together to focus on delivering the best results for the system as a whole, without needing to be coordinated by management who take away their autonomy.

You shouldn't have to rely on serendipitous lunch time discussions to propagate key insights that lead to important redesigns of your organizational and technical boundaries. Use established practices like show and tells and cross-team pairing to ensure knowledge travels and networks form in your organization.

Show and tells

At the end of every iteration, each team should demonstrate what they have been working on, with an open invitation for anyone else in the organization to attend. Teams will have a clear understanding of the key priorities other teams are working to, the user needs they are solving, and the technical implementation of their system.

Cross-team pairing

Moving software engineers between teams for short periods is an effective way for deep knowledge transfer that cannot be shared in show and tells. It can also be an effective approach for other roles, including testers, product managers, and user researchers. Cross-team pairing involves pairing up with someone from another team, working on real items, and contributing to the end result.

Cross-skill pairing

Cross-skill pairing involves two people with different skill sets pairing together on real work items. Cross-skill pairing can lead to valuable insights that only arise through combining the knowledge and thinking patterns of multiple disciplines. For example, when product managers pair up with software developers and better understand how the technology works, they may produce more innovative and practical solutions.

Mapping the System

Mapping out your system is an ideal way to improve visibility and understanding of the system as a whole. Techniques such as value stream mapping and event storming have proved hugely successful in mapping out systems. To understand value stream maps, it's first important to understand the Theory of Constraints (ToC).

Theory of Constraints

> *Since the strength of the chain is determined by the weakest link, then the first step to improve an organization must be to identify the weakest link.*
>
> —Eliyahu M. Goldratt

In his best-selling classic *The Goal*, Eliyahu Goldratt demonstrates how bottlenecks can be the biggest impediment to organizational performance. If two teams depend on a third team, it doesn't matter how fast the first two teams can create value; the performance of the organization will be limited to the rate at which the third team can feed the other two teams. This means that you have to design systems to avoid bottlenecks. To do so, you must first identify where bottlenecks exist between teams and try to eliminate them by redesigning boundaries in teams and code.

Goldratt's book is inspirational reading. It was originally developed based on experiences in the manufacturing industry, but is highly relevant to software having influenced many approaches, including DevOps and continuous delivery.

Value Stream Maps

One practical application of ToC is value stream mapping. Essentially, value stream maps line up each step in the process from customer demand to business value and show how much time is spent on each (Figure 5-3). With the information presented clearly and visually, it becomes possible to measure bottlenecks and understand how they are affecting your core domains and value propositions, providing insights to better design teams and code.

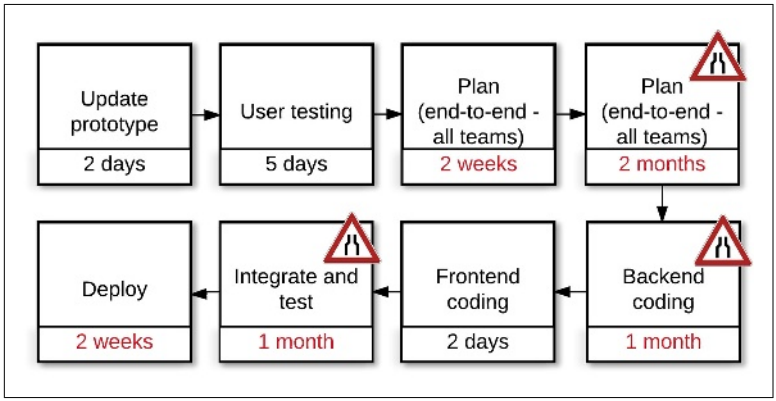

Figure 5-3. An example value stream map

Big Picture Event Storming

Event storming brings together a diverse mix of people from across a company where they map out full system processes as a sequence of events linked to customers, external systems, and triggers.

Participants of event storming workshops use sticky notes to map out events running along large stretches of wall (see Figure 5-4). According to Alberto Brandolini, the inventor of event storming, you need unlimited modeling space.

By bringing together representatives from across the business and mapping out an entire system, the insights produced by event storming workshops can be invaluable in designing effective organizational and technical boundaries.

Figure 5-4. An event storming workshop in progress

Event storming can work in both new and existing systems. It can be used to kick off new initiatives by exploring new domains and systems, or it can be used to map out existing domains and systems. To learn how to run event storming workshops, check out Alberto Brandolini's book *Event Storming*.

System Validation and Optimization

One of the most frustrating aspects of designing boundaries is the lack of quantitative validation. There is no direct metric or measure that proves your proposed or existing design is optimal or effective. It makes it challenging to convince others that your design is in the best interests of the business. However, creating autonomy is about grouping things that change together. There are techniques you can use to demonstrate how effective your designs are based on things that change together.

Historical Work Item Analysis

When modifying an existing system or commencing a rebuild, you should have a rich history of historic data in your work tracking tool. By replaying past work items against your proposed new design, you can get an idea of how effective your new design may be.

Historical work analysis involves reviewing work items that have been completed, usually over the past 6–12 months, and measuring how many teams would need to collaborate to complete the piece of work. Your ideal scenario is that the highest amount of high-value work is owned by a single team who have the autonomy to implement the end-to-end feature without being blocked by other teams.

Domain Evolution

In most domains, some parts of the domain change more frequently than others. There are stable boundaries that remain largely unchanged over time, whereas others are far more fluid. Areas of the domain likely to remain stable are indicators of future autonomy. Conversely, it would be risky to optimize too strongly for autonomy in areas of a domain known to be volatile—if volatility continues, boundaries will need to constantly readapt, and collaboration costs will be high.

At NDC London in late 2014, Udi Dahan spoke about the challenges he faced trying to identify optimal service boundaries for a large healthcare provider. During the talk he cites examples of the challenges outlined throughout this report—such as how to decide between the multitude of conflicting heuristics for finding bounded contexts. Toward the end of the talk, Udi explains how analyzing the history of the well-established healthcare domain was a crucial factor in the boundaries he chose. He demonstrates how he aligned bounded contexts with clinical contexts based on how the domain had changed over the years.

Lagging Validation

Lagging validation approaches inform you after the fact. Lagging validation metrics are usually key performance indicators (KPIs) or business metrics. They tell you about the overall health of the business (number of new sign-ups, trades per second, etc.). If you change your boundaries and your metrics show an improvement, that's an encouraging sign, but not a guarantee you have made good choices. To learn more about business metrics and understanding how to measure the performance of your organization, Alistair Croll and Benjamin Yoskovitz's *Lean Analytics* (O'Reilly, 2013) is highly recommended reading.

Complexity Theory

Theories of complexity help you reason about systems. One theory is Cynefin, a decision framework for analyzing your domain and choosing the appropriate strategy moving forward. For example, should you hire experts or adopt an experimental mindset to find a solution for yourself? Cynefin helps you to classify your problem space into one of five domains:

Simple
: There is predictability between cause and effects. The solution is largely obvious and existing best practices.

Complicated
: A cause-and-effect relationship exists, but is less obvious. There is no best practice, but multiple good practices. You may want to hire an expert in this area to help you proceed.

Complex
: There is no cause-and-effect relationship. The system can be modified by agents operating within it. The best solution is to take an experimental approach instead of trying to design up front. Emergent properties.

Chaotic
: Cause and effect are unclear. Due to the extreme uncertainty, any practice will be novel. The focus will be in innovating in new areas or stabilizing out-of-control situations. Trying to follow best practice here would be foolish.

Disorder
: Lack of clarity determining which domain is relevant.

To learn more about Cynefin, the best place to begin is Cognitive Edge's website. Another popular approach is Promise Theory. To learn more, check out Mark Burgess's *Promise Theory* (CreateSpace, 2014).

CHAPTER 6

Architecting Autonomous Applications

> *The point of microservices is to unblock independent queues of work. Both in the system of services and the system of people.*
>
> —Andrew Clay-Shafer

Organizations cannot achieve high autonomy if there are couplings in the software. Couplings in software will result in couplings between teams. Consequently, all of the rich domain and business knowledge has to be taken into account when architecting boundaries. Software architecture should be a collaborative activity involving not only technical people, but a variety of stakeholders from product managers, to user experience designers, to business analysts. So regardless of your skill set and job title, software architecture is relevant to you.

Making Software Architecture Cross-Functional

To make software architecture a more cross-functional activity that includes a diverse range of people, start with the obvious: if you're a developer/architect, invite others to your sessions, share your diagrams, and make sure they at least have access to the information and an opportunity to contribute. Ask them for their opinion, and encourage them to have an opinion. Obviously, remind them there are no stupid questions. If you're on the other side of the fence (i.e.,

if you're not involved in architecture), try showing an interest. Ask if you can come attend workshops and view the diagrams.

The biggest hurdle to achieving a more cross-functional influence on architecture is clarifying details. If nontechnologists are overwhelmed with complicated information that isn't relevant to them (e.g., diagrams of network switches, reverse proxies, and wire protocols), they will more than likely be overly confused and not want to contribute. So how do you decide which information should be collaborative and which should be exclusive to engineering teams?

Some business-minded people may want to learn more about network switches, reverse proxies, and wire protocols. In fact, it may be highly relevant to understanding how products work in some organizations.

Whiteboards and diagrams go a long way to making software architecture a more collaborative activity, when done right. If you stay at the right level of detail, instead of creating in-depth visualizations that are too confusing, you stand a better chance of getting your point across as a technologist. Language, again, is a vital factor. If your diagrams align with business concepts (i.e., if you've named your software services and components after the domain concepts and business processes they represent), the information you are presenting should naturally make sense to non-IT people.

If you're not sure how to convey your software architecture, start at the highest level and drill down into the details as necessary. Start with a system context diagram, and then drill down into domain use case diagrams and domain stories. To learn more, check out Simon Brown's C4 model of diagramming.

Microservices

Microservices enable you to create autonomous teams because they create autonomy in your code. Since the early 2010s, microservices have emerged as a popular, if not pop culture, approach to building large-scale software applications by breaking the system down into many small pieces.

The "micro" in microservices is hotly debated and unlikely to ever have a clear, consistent definition. But don't worry about that, just

think of microservices as a way to create autonomous teams, by creating autonomous software architecture. The real challenge of microservices you should focus on is getting your boundaries right. If you get your boundaries wrong, you will introduce bottlenecks and coupling between teams, who will then lack autonomy to continuously discover and deliver.

So what does a microservice architecture look like? Consider the example of an ecommerce site. Prior to microservices, it may be a single codebase that is deployed together as a single unit, just one software application, but with potentially multiple teams owning it. This pattern is typically known as The Messy Monolith or Big Ball of Mud. See Figure 6-1.

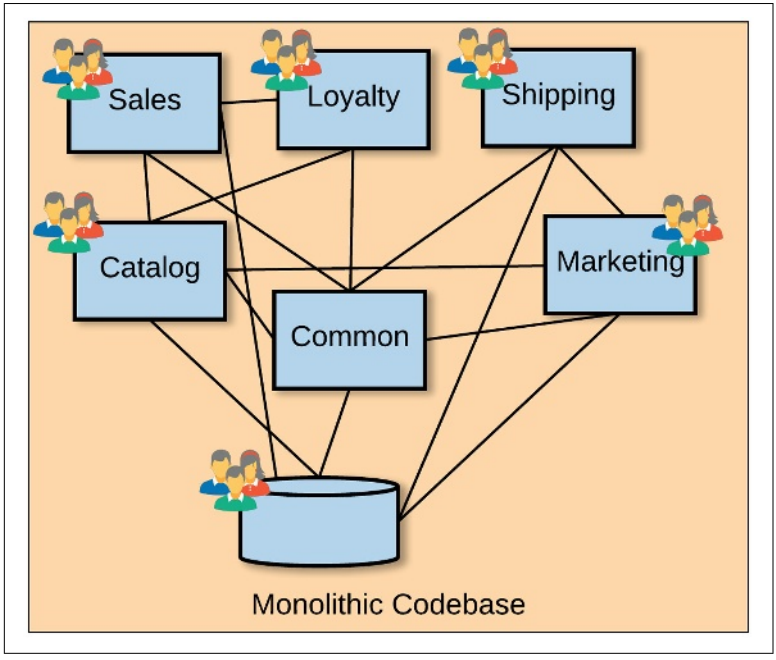

Figure 6-1. The Messy Monolith

Of course, monoliths are susceptible to fairly obvious problems. With multiple teams all working in the same codebase, they are likely to interfere with each other. For example, one team may be ready to release a feature, but another team may have made some changes that have introduced a bug, thus delaying the release. Subsequently, teams become bottlenecked by other teams and they are

forced to follow certain conventions around technologies, tooling, release processes, and so on.

Microservices addresses the coupling drawbacks associated with monoliths by breaking up the monolith into smaller pieces. If you've done microservices right, these smaller pieces will allow each team to choose their own technologies and tools (though some standardization is sensible). So converting the ecommerce application to microservices may result in the monolith being broken down into a catalog microservice, an orders microservice, a checkout microservice, and so on. Microservices doesn't have an opinion on where you draw the boundaries, although most people in the community do, and they often point to bounded contexts as shown in Figure 6-2.

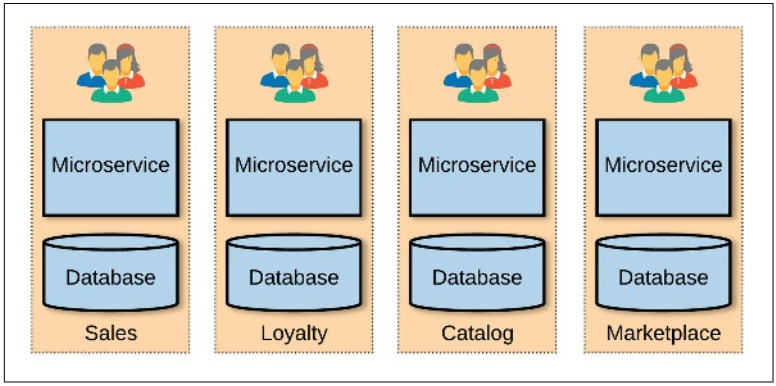

Figure 6-2. Aligning microservices and bounded contexts

Microservices and Bounded Contexts

Should there be a 1:1 mapping between microservices and bounded contexts? In an ideal world, yes. It would make life easier if each of your bounded contexts could be built as a single microservice. In practice, you'll often find one bounded context may contain multiple runnable applications. However, the most important thing to focus on is team autonomy. If a team owns a bounded context, they will own the things that change together whether they are spread across one or multiple microservices.

No sharing between contexts

Arguably the most important rule of microservices and bounded contexts is no sharing. Each team should have its own code libraries and databases. Sharing code and databases introduces dependencies

between teams (Figure 6-3). Team A might introduce a breaking change that breaks Team B's code. As a consequence, the teams will decide that higher collaboration is required around the shared resource. The additional collaboration then results in a higher cost of change, slower rate of change, and lots of wasted time. All of these pains can significantly reduce a team's autonomy. You can avoid them by being highly averse to sharing.

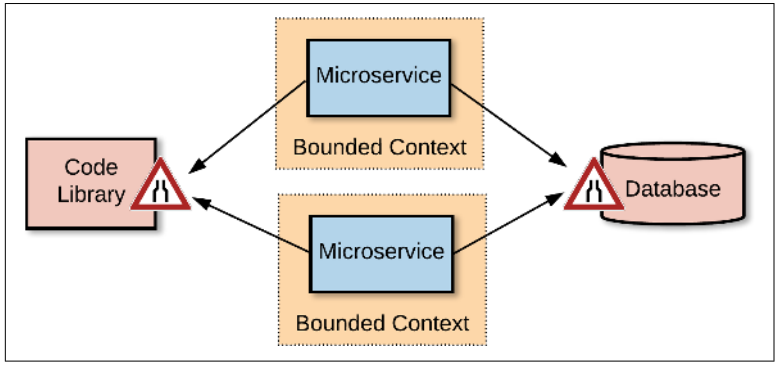

Figure 6-3. Avoid sharing databases and code between contexts

Do bounded contexts include the UI?

A common point of contention is whether bounded contexts own UI. To sidestep the frivolous debate, think in terms of autonomous contexts and the confusion goes away—the goal is to encapsulate whatever is necessary for a team to be autonomous. If the UI and business logic are owned by different teams, a bottleneck will be introduced, reducing team autonomy. Therefore autonomous contexts can own business logic and UI.

In any case, not all autonomy and bounded contexts will own any UI at all. Some contexts may expose UIs and APIs, whereas some may expose just UI, or just APIs, especially as the API-first and platform thinking movements continue to grow. The proliferation of integrations between disparate software systems and digital products is all made possible thanks to APIs.

A big challenge for bounded contexts that do own the UI is owning the UI across many different devices. Consider a product with a web UI, IOS app, Android app, smart TV app, and so on. Is it realistic or even possible for all of these UIs constructed from different technol-

ogies to be owned by a single autonomous team? This challenge is discussed later in the chapter.

Event-Driven Microservices

Coupling at the technology level has many guises, from platform coupling to database coupling. Event-driven architecture is an architectural pattern that can be applied to reduce some forms of coupling at the technical level, particularly temporal coupling, where one service must immediately respond to another.

Consider the scenario of a concert ticket booking website. Imagine a surge in traffic just as tickets become available for Adele's latest concert, and the payment processing system goes down under the extreme load. If the Sales context had a temporal coupling to the Payments context, the Sales context would receive an error from the Payments context and no tickets could be sold. With event-driven architecture, though, the Sales context would publish events and the Payments context would subscribe to them. If the Payments service did go down for any reason, it would be able to pick up all of the events once it comes back to life, meaning no lost revenue for the business.

For more information on event-driven architecture, you can download the free O'Reilly ebook *Software Architecture Patterns* by renowned architect Mark Richards.

Composing Applications

No matter how autonomous contexts are, they must collaborate together to form complete systems and fulfill full user journeys. There are a variety of established patterns you can use to compose your systems and user journeys of multiple contexts while mitigating loss of team autonomy. You'll notice these patterns are aimed at decoupling at the UI level. This is usually where the majority of organizations struggle the most with microservices. They understand the value in moving away from the shared database and the monolithic codebase, but they don't realize that a monolithic frontend owned by a single team introduces significant bottlenecks and reduces a lot of autonomy.

Composite User Journeys

One of the least painful ways to decouple contexts at the UI level is for each individual web page to be fully owned by a single context. To the user, the experience should not suffer. As they click links and fill in forms, being redirected to different pages in the user journey along the way (Figure 6-4), they view a range of pages supplied by different contexts, while still enjoying the same consistent user experience.

Consider the example of the government agency discussed previously in Chapter 1. Citizens go on a journey through a Review, Resubmit, and Renegotiate process, and each of those business process steps is a context. Accordingly, Review pages would be provided by the Review context, Resubmit pages by the Resubmit context, and so on. Each context owns its UI, so whenever the team owning the context implements a new change, they implement the UI work, the backend work, and database work without being blocked or needing help from other teams.

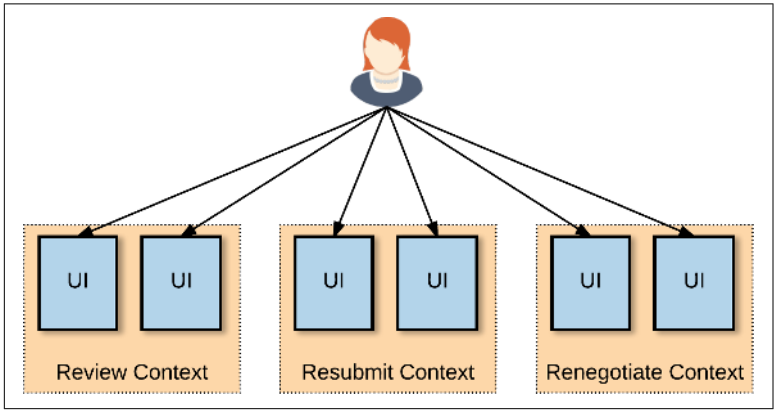

Figure 6-4. A composite user journey

Composing user journeys from multiple bounded contexts does introduce challenges around consistency. To provide a consistent look and feel, teams must use the same UX patterns and themes. Clearly, this pattern introduces the need for greater discipline and self-organization, but the reward is greater autonomy.

Composite User Interfaces

For some scenarios, you'll find it's not possible for a single page to be owned by a single context. The page may require data from multiple different contexts. To solve this problem, consider creating composite UIs. With composite UIs, individual pages are broken down into HTML fragments that are supplied by different contexts (Figure 6-5). For example, consider Amazon's product details page, which contains a product description, price, recommendations, shipping predictions, and more. Each section of this page could be an HTML fragment fed by a different bounded context, avoiding the need to create a monolithic UI where every change would need to span multiple teams.

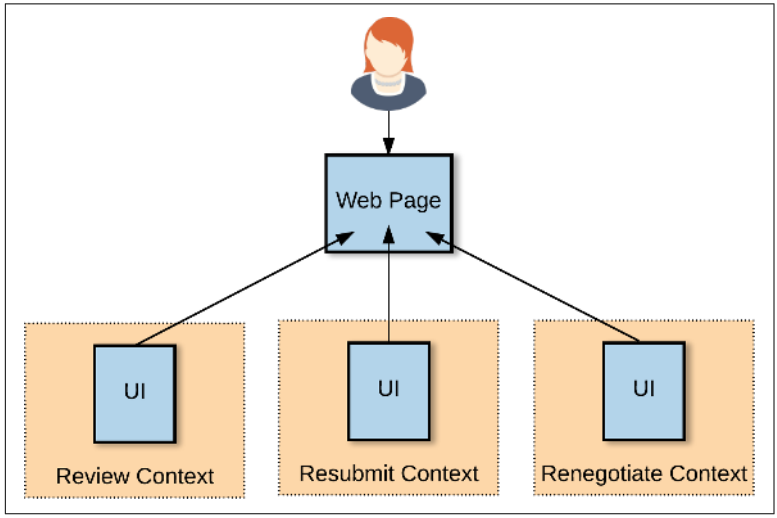

Figure 6-5. A composite user interface

UI composition also introduces significant challenges around consistency and coordination, even more so than composite user journeys due to the additional complexity of decomposing an individual page. Many organizations are successfully applying this approach, and a number of frameworks to support the pattern have been created. A good place to start if you are keen to learn more is Open Table's Open Components. You should also check out Jimmy Bogard's talk from NDC Oslo 2017: "Composite UIs: The Microservices Last Mile".

Backends for Frontends (BFFs)

There are many valid reasons why you should consider creating a dedicated frontend team. It's been argued so far that chopping up the UI and distributing it among bounded contexts is the optimal strategy. But remember, autonomy is about finding things that change together. Sometimes the things that change together are the UI pieces. In such cases, it makes sense to create dedicated/monolithic frontend applications owned by a single team. The frontend web applications will have their own API layer, known as a Backend for Frontend (BFF) (Figure 6-6).

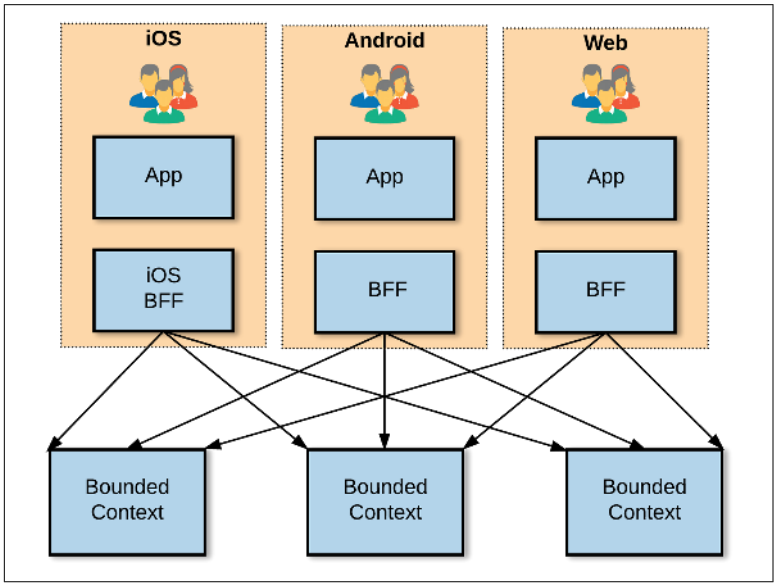

Figure 6-6. BFFs—Backends for Frontends

Most often, BFFs are used when a product spans multiple devices (e.g., the iOS app, the Android app, the web app), where collectively it's simply too much for a single team to manage. But it's never an easy decision. The best you can do is focus on the goal of creating the most autonomy in the highest value places. Use the techniques presented in previous chapters to help you understand what is valuable to the business, and which things change together.

To learn more about BFFs, Sam Newman's detailed post on the topic is a quality read.

Brownfield Strategies

Talking about microservices and drawing pretty diagrams of fancy autonomous services is easy, but for many teams the reality is a large, tightly coupled codebase that cannot easily be broken down into microservices. What should those teams do?

A number of strategies exist for transitioning to autonomous architectures in even the most challenging brownfield scenarios. As you saw in previous chapters, one approach is to slowly break out the highest value areas. Another approach is to instrument your system and collect metrics so that you can analyze dependencies between different components. You can then use knowledge of dependencies to inform your plan of attack—for example based on the component with the fewest dependencies.

Whatever scenario you find yourself in, understanding the business goals, focusing on what is core to the organization, and analyzing your domain for cohesion will all give you the rich knowledge you need to devise an effective transition strategy.

About the Authors

Nick Tune is a strategic technical leader who aligns organizations and software systems by starting with customer needs, focusing on a clear vision of product strategy, and including everyone in a continuous design and delivery process. He is the coauthor of *Patterns, Principles, and Practices of Domain-Driven Design*, an international conference speaker, and a principal engineer at Salesforce. He blogs at *ntcoding.co.uk* and tweets from @ntcoding.

Scott Millett is the director of IT for Iglu.com, and has been working with .NET since version 1.0. He was awarded the ASP.NET MVP in 2010 and 2011, and is the coauthor of *Patterns, Principles, and Practices of Domain-Driven Design* and author of *Professional ASP.NET Design Patterns* and *Professional Enterprise .NET*. Feel free to drop him a line on Twitter (@ScottMillett) or via email (*Scott@elbandit.co.uk*).