
INNOQ
Modern Domain-driven Design
From the business model to systems – agile and socio-technical
Michael Plöd

Modern Domain-driven Design

From the business model to systems – agile and socio-technical

Michael Plöd

innoQ Deutschland GmbH
Krischerstraße 100 · 40789 Monheim am Rhein · Germany
Phone +49 2173 33660 · www.INNOQ.com

Layout: Tammo van Lessen with XeLaTeX
Design: Murat Akgöz
Typesetting: André Deuerling

Modern Domain-driven Design – From the business model to systems – agile and socio-technical
Published by innoQ Deutschland GmbH
1st edition · January 2026

Contents

1 Introduction

Domain-Driven Design (DDD) has been a distinctive approach to designing complex software systems since the publication of Eric Evans' book in 2003. Since then a lot has changed, both within the DDD community as well as in related disciplines such as agile product development, organizational design, and modern software architecture.
This primer is intended for anyone working in this field:

- **Software developers and architects** looking for practical patterns for robust systems.
- **Agile coaches, product owners, and business analysts** who want to understand how DDD can be used as a bridge between business and technology.
- **Engineering leads and executives** who operate at the interface between organization, architecture, and strategy.

1.1 Why a comprehensive overview?

In recent years, new methods and approaches have emerged that complement and extend the original core of Domain-Driven Design:

- **Collaborative modeling** techniques such as EventStorming, Domain Storytelling, and Event Modeling have demonstrated the value of bridging the gap between business and technology.
- **Strategic DDD** has gained massive importance and has been taken up by tools such as the Bounded Context Design Canvas and concepts from *Team Topologies*.
- **Tactical DDD** is now thought of in a broader architectural context, from classic monoliths to microservices to event-driven architectures and cloud-native systems.

At the same time, the world of work has changed: Agile methods have become mainstream, product organizations are working more in a cross-functional manner, and topics such as *socio-technical architectures* and *fast flow principles* are gaining in significance.

1.2 Overview of the contents of the primer

This primer is divided into several parts, each of which highlights different facets of modern Domain-Driven Design:

DDD as a way of working and an attitude An introduction to the central ideas of Domain-Driven Design. Throughout this chapter, it becomes clear that DDD is not just a technical pattern set, but a way of thinking and working that correlates strongly with the principles of agile development: close collaboration, iterative approaches, and a focus on value creation. Misconceptions are clarified and initial guidelines for use in modern product and organizational environments are set.

Guiding approaches in the DDD environment Teams need guidance to move from an attitude and mindset to concrete practices This chapter shows how DDD becomes tangible through two complementary approaches: Eric Evans' Model Exploration Whirlpool illustrates the iterative nature of modeling as a learning process, while the DDD Crew's DDD Starter Modeling Process provides a pragmatic step-by-step framework for getting started. Together, they help to structure uncertainty and pave the way for the following methods in the primer.

Collaborative Modeling This section deals with methods for people with domain and software engineering knowledge to work together on a model. EventStorming, Domain Storytelling, and Event Modeling help to create a shared understanding and make implicit knowledge visible. These techniques promote open communication, facilitate requirements engineering in agile environments, and form the basis for viable models.

Strategic DDD The focus here is on structuring complex domains. Concepts such as bounded contexts, subdomains, and context maps help to clarify responsibilities and interfaces. Supplemented by tools such as the Bounded Context Design Canvas or Core Domain Charts and concepts from *Team Topologies*, it shows how to design technical and organizational boundaries in such a way that value streams are optimally supported.

Tactical DDD This part is dedicated to the specific patterns and architectural possibilities that are derived from the technical models. Aggregates, Value Objects,

Entities, Repositories, and Domain Events provide the basic building blocks for robust solutions. This is complemented by architectural approaches ranging from monoliths to microservices to event-driven architectures, as well as patterns such as Ports & Adapters, CQRS, and Event Sourcing.

Conclusion: Do's and Don'ts in Domain-Driven Design Domain-Driven Design is not a dogma, but rather an attitude that shapes thinking, collaboration, and architectural decisions. Nevertheless, there are typical pitfalls that almost every team experiences at some point, as well as typical success factors that determine whether a project will succeed. This concluding chapter summarizes the most important do's and don'ts. Not as a checklist, but as a guide for practical application.

1.3 Purpose of this primer

This primer is intended to provide a **compact but comprehensive overview** of modern Domain-Driven Design:

- It explains the multitude of concepts and methods.
- It shows how DDD is used in organizations today, not only in code, but also in collaborative work.
- It highlights pitfalls and misunderstandings that occur repeatedly in practice.

My goal is not to shed light on every detail. Rather, I want to provide guidance and clarify how Domain-Driven Design has evolved in recent years. For those who want to dig deeper, there are references to further resources at the end of the primer.

2 DDD as a way of working and a mindset

2.1 Introduction

Domain-Driven Design (DDD) is often perceived as a collection of methods, patterns, and techniques. In public discussion, the focus often tends to be on **artifacts** such as *Bounded Contexts* or *Aggregates*. Other architectural patterns such as *Hexagonal Architecture* or *Microservices* are often mentioned in the same breath, but these are not DDD-specific artifacts per se, rather related approaches that can be combined well with DDD. The real core of Domain-Driven Design does not lie primarily in these artifacts, but in the **path** that leads to them: *How do I identify Bounded Contexts? How do I slice domains? Why do I choose a particular architectural pattern, or consciously decide against it?* It is precisely this process of understanding, discussing, and deciding that defines Domain-Driven Design. **It's about the journey, not the destination.** This chapter shows that DDD is not just a toolbox for architects and developers, but a **way of working and an attitude** that shapes collaboration, ways of thinking, and decision-making processes. It also highlights the close connection to agile principles.

2.2 Domain-Driven Design as a way of working

DDD is not a methodology that you introduce once and then apply like a rigid set of rules. Rather, it is a **continuous practice** that focuses on understanding the domain and making that understanding the foundation of technical design. Key features of this way of working are:

- **Iterative approach**: Knowledge about the domain grows over time. DDD emphasizes that models and implementations mature together with this understanding.

- **Close collaboration**: Subject matter experts and developers do not work separately, but in an ongoing joint dialogue.
- **Experimentation and learning**: Hypotheses about the domain are tested, adapted, or discarded early on. This also involves transforming the implicit mental models of those involved into **explicit, shared knowledge**. Alberto Brandolini, the inventor of EventStorming, put it aptly by asking: What actually goes into production — the knowledge of the domain experts or the hypotheses and assumptions that exist in the minds of the developers about this knowledge? It is precisely this process of visualization and comparison that forms the core of learning in the DDD context. These aspects align DDD with agile values as described in the **Agile Manifesto**: "Individuals and interactions over processes and tools. Working software over comprehensive documentation. Customer collaboration over contract negotiation. Responding to change over following a plan"

2.3 Attitude: The mindset behind Domain-Driven Design

In addition to the way of working, DDD emphasizes a particular **attitude** that affects several dimensions: 1. **Respect for domain expertise**: DDD puts the domain at the center. It is not about using technology for its own sake, but about precisely understanding and modeling the business and application logic. 2. **Ubiquitous language**: The language of the domain becomes the language of implementation. This requires openness, empathy, and the willingness to build bridges between different disciplines. 3. **Long-term value orientation**: Instead of focusing on short-term optimizations, DDD pursues the goal of developing sustainable models and architectures. 4. **Acceptance of complexity**: DDD is not a tool for eliminating complexity, but for making it **visible and manageable**. 5. **Continuous learning**: DDD thrives on the fact that models cannot be perfect from the very beginning. Iteration 1 can therefore often only claim to have been "always very diligent." A truly sustainable model can only be achieved through continuous learning, reflection, and joint development. This attitude closely links DDD with agile principles, in which iterative approaches and continuous improvement are crucial.

2.4 The relationship to agility

DDD cannot be viewed in isolation from agile principles. Rather, both approaches reinforce each other. There are numerous similarities to DDD, especially in the **Agile Manifesto**:

- **Business people and developers must work together daily throughout the project**: This principle is specifically embodied in DDD, where domain experts and developers work together to design models and continuously adapt them. Ubiquitous Language and collaborative workshops are direct manifestations of this principle.
- **Feedback cycles**: Agile methods rely on short iterations in which feedback is obtained. DDD concretizes this approach by demanding feedback not only at the technical level, but also at the domain model level.
- **Collaboration**: Agility emphasizes cross-functional teams. DDD makes this idea explicit through all the ideas evolving around knowledge cruching and collaborative modeling approaches.
- **Adaptability**: Agility aims to be able to respond to change. DDD offers mechanisms such as **Bounded Contexts** and **Context Maps** to incorporate these changes into the architecture in a structured way.

2.5 Practical implications

DDD as a way of working and an attitude manifests itself in numerous facets in everyday life:

2.5.1 Workshops and collaborative modeling

Formats such as **Event Storming** or **Domain Storytelling** illustrate that DDD relies heavily on visual, collaborative, and interactive methods. This results in models that are understandable to both business and technical experts.

2.5.2 Architectural decisions

The DDD approach means that architecture is not created in isolation on the "drawing board," but grows out of the domain-specific knowledge. Decisions such as the division into Bounded Contexts are the result of domain-specific boundaries.

2.5.3 Prioritization and value creation

DDD sharpens the focus on which parts of a domain are **strategically relevant**. Instead of treating everything equally, DDD distinguishes between core (sub)domains, supporting subdomains, and generic subdomains. Investment and staffing decisions are derived from this.

2.5.4 Continuous refactoring

The mindset behind DDD accepts that models are never perfect. Instead, the aim is to continuously improve them as new knowledge becomes available.

2.6 Challenges in its practical application

Embracing DDD as an attitude is no trivial matter. One of the biggest hurdles is **silo thinking**: when departments and IT operate in strict isolation from one another, there is no basis for a common language and the necessary ongoing dialogue.

Equally challenging is the ever-present **time pressure**. In projects with tight deadlines, careful modeling is often considered a "luxury," even though it is crucial for the long-term quality and sustainability of the systems. In addition, there are numerous **misconceptions** about Domain-Driven Design in the field. Dogmatic views on Domain-Driven Design are often encountered. For example, some believe that seriously implementing DDD inevitably leads to a microservice architecture. Others argue that DDD requires a hexagonal architecture or automatically leads to event-driven systems. These views are simplistic and incorrect. DDD does not prescribe a specific target architecture, but rather provides thinking

tools and practices for making the right decisions for a specific domain and its context. This is precisely where the real strength of DDD lies: it is about the path to knowledge and conscious choice, not about ready-made dogmatic solutions.

Another point that is often overlooked is the **scope of adoption of domain-driven design**. In its fully developed form, DDD is time-consuming and therefore also cost-intensive. It would be inefficient to apply this approach equally to every subdomain or every part of a system. Instead, DDD clearly states that the methods and practices are most useful when it comes to **core domains**: those areas that are crucial for strategic, economic, or business success in the medium to long term. In supporting or generic subdomains, however, it is often sufficient to choose more simple approaches.

2.7 Opportunities and added value

The opportunities and added value of Domain-Driven Design lie less in general buzzwords such as communication or sustainability and more in the very concrete effects that arise from the close connection between domain expertise and technical implementation.

A key added value is **more maintainable systems**. Because the structure of the code arises directly from domain thinking, the software becomes clearer, more comprehensible, and thus more robust. Functional changes can be more easily transferred to the technical world without causing fractures. This reduces the risk of systems becoming unmanageable. Another major advantage lies in **alignment**: DDD promotes the alignment of functional boundaries with the vertical modules of the software architecture. Bounded Contexts thus become more than just an architectural construct; they also help to draw clear deployment boundaries and clearly define responsibilities. This alignment can also be transferred to organizational aspects. If we consider the ideas from *Team Topologies*, for example, Bounded Contexts are an excellent starting point for tailoring teams. This results in teams whose responsibilities correspond to the functional boundaries and who can work autonomously.

DDD not only creates technically clean architectures, but also helps to harmonize organization, architecture, and domain. This combination of technology and organization is a specific and strategically relevant added value that sets DDD apart from many other approaches.

2.8 Conclusion

DDD as a way of working and an attitude is much more than a collection of methods and patterns. It combines technical craftsmanship with an attitude that focuses on **collaboration, respect for the domain, and conscious engagement with complexity**. In combination with agile principles and the values of the **Agile Manifesto**, this creates a powerful foundation for designing complex software systems in a sustainable way.

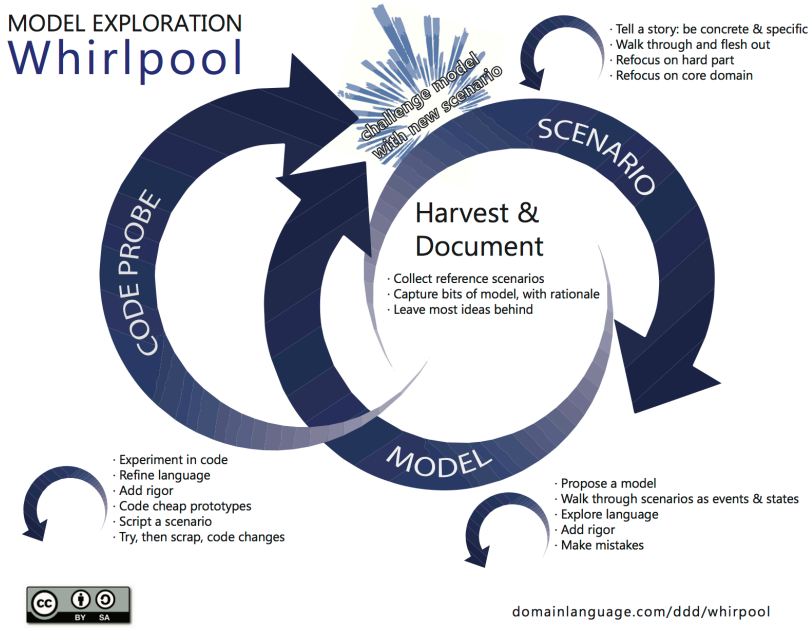
3 Guiding approaches in the DDD environment

In the first part of this primer, we saw that Domain-Driven Design is much more than a toolbox of patterns and artifacts. It is an approach that focuses on collaboration, respect for expertise, and continuous learning. But this immediately raises a specific question for many teams: *How exactly should we get started?* This is where guiding approaches for domain modeling come into play. They help to make the abstract attitude tangible and translate it into concrete first steps. It is important to note that DDD does not prescribe a rigid methodology that must be followed from A to Z. Rather, it is about showing ways that give teams orientation without forcing them into a tight corset. In this chapter, we present two approaches that have gained particular significance in the community: Eric Evans' **Model Exploration Whirlpool** and the DDD Crew's **DDD Starter Modeling Process**. They are helpful in different ways: The whirlpool describes the thinking and learning movements that naturally arise during modeling, while the starter process offers a pragmatic collection of specific activities that is particularly suitable for getting started in a hands-on manner. Together, they provide a valuable reference to apply Domain-Driven Design in your day-to-day work.

3.1 Modeling as a cycle: The Model Exploration Whirlpool

Eric Evans, the founder of Domain-Driven Design, recognized early on that modeling domain expertise is not a linear process. Gathering requirements, designing a model, and then implementing it without change, that's not how it works in reality. Instead, teams move in circles, refining ideas, discarding hypotheses, and learning to better understand the domain step by step. To describe this process, Evans coined the metaphor of the **Model Exploration Whirlpool**. The whirlpool represents a constant cycle between different activities: conducting technical discussions, setting up models, implementing prototypes, redrawing boundaries, and testing hypotheses. There is no fixed beginning and no definitive end. The

current of the whirlpool repeatedly pulls the participants back in and ensures that the model matures a little more with each turn.



(Image source: https://domainlanguage.com/ddd/whirlpool)
This metaphor illustrates several key insights:

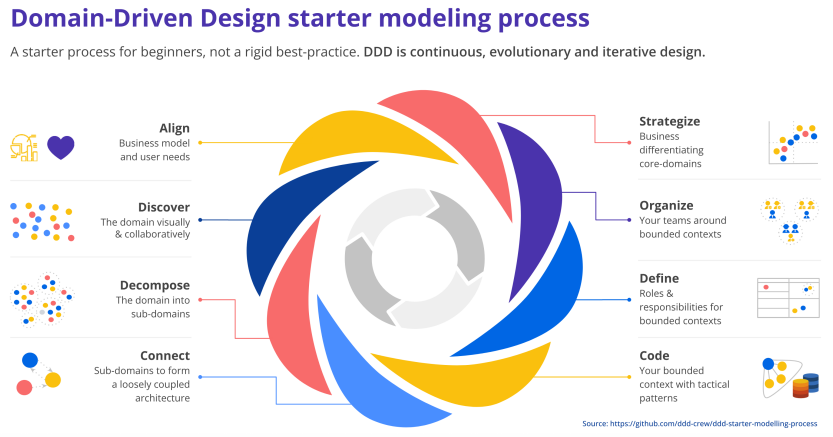
- **Iterative learning is inevitable.** A model that is "finished" on the first attempt does not exist in practice.
- **Expertise and technology are interdependent.** Insights from implementation flow back into the model, just as technical discussions have a direct impact on the code.
- **Uncertainty is part of the process.** Instead of being discouraged by vague or contradictory requirements, the whirlpool embraces this ambiguity as the norm.

For teams, the Model Exploration Whirlpool is an invitation to cultivate **serenity in dealing with complexity**. It conveys the conviction that it is not about finding

the perfect solution from the very beginning, but that learning and adapting are integral parts of domain modeling.

3.2 Getting started step by step: The DDD Starter Modeling Process

While the whirlpool provides a helpful metaphor, teams often need something more tangible in practice. Especially when getting started, it is important to have a common guideline that provides orientation. This is where the DDD Crew's **DDD Starter Modeling Process** comes in. This process offers a **lightweight sequence of steps** that allows even less experienced teams to organize their first modeling sessions. It should not be understood as a strict set of procedural steps, but rather as a pragmatic collection of recommendations that have proven their worth in many projects.



(Image source: https://github.com/ddd-crew/ddd-starter-modelling-process)

Typical elements of this process are:

- **Create a common understanding of the domain.** This often takes the form of an exploration workshop, using EventStorming or similar techniques. The goal is to make implicit knowledge visible and establish a common language.

- **Identifying initial subdomains and bounded contexts.** Instead of capturing the entire complexity at once, subdomains are identified and given responsibilities.
- **Developing the ubiquitous language.** Terms that arise in discussions are clarified and used consistently in the models and in the code.
- **Refine step by step.** The process encourages starting with simple visualizations or prototypes and expanding them as needed.

What is particularly valuable about the Starter Modeling Process is its **pragmatism**: it is clear enough to provide guidance, but open enough to be adapted to different contexts. Teams that have no previous experience with DDD can follow it and achieve initial success without knowing in detail what the entire puzzle will look like in the end.

3.3 Whirlpool and Starter Process in interaction

At first glance, Whirlpool and Starter Process might seem like opposites: one describes an open, nonlinear circle, the other a specific collection of activities. In reality, however, they complement each other perfectly. The **Model Exploration Whirlpool** provides the **philosophy**: modeling is not a linear process, but an iterative learning game that is never complete. It sensitizes teams to the fact that setbacks and loops are not mistakes, but the norm. The **DDD Starter Modeling Process** provides the **practice**: it gives teams a concrete roadmap to guide them, especially in the early sessions when uncertainty is high. Taken together, this creates a **dual frame of reference**: the whirlpool explains why Domain-Driven Design works the way it does. The Starter Process shows how to take action in everyday life.

3.4 Outlook: Our point of reference in the rest of the primer

In the rest of this primer – in the chapters on **Collaborative Modeling**, **Strategic DDD**, and **Tactical DDD** – we will refer to the **DDD Starter Modeling Process**. It serves as a common thread because it is well suited to putting methods such as EventStorming, Domain Storytelling, and Context Mapping into a comprehensible sequence. At the same time, the Model Exploration Whirlpool remains an important background idea. No matter which technique or tool is used, thinking in iterative learning cycles is the foundation for DDD to be impactful.

3.5 Conclusion

In this chapter, we bridged the gap between the fundamentals and specific practices. We learned about two approaches that offer different but complementary perspectives. The Model Exploration Whirlpool reminds us that DDD is an ongoing learning process. The Starter Modeling Process gives us a manageable approach to getting started with this process. Together, they form the frame of reference with which we will continue to work in the coming chapters, whether in collaborative modeling, strategic considerations, or tactical design. Anyone who wants to understand and apply Domain-Driven Design needs both: confidence in the whirlpool of learning and the courage to take clear first steps.

4 Collaborative Modeling

In the first part of this primer, you saw that Domain-Driven Design is not just a collection of methods, but a way of working and a mindset that focuses on collaboration, respect for domain expertise, and continuous learning. This is where the comparatively young discipline of **collaborative modeling** comes in: it makes these principles tangible in practice by being highly interactive and having no significant barriers to entry.

4.1 Introduction & Principles

Collaborative modeling describes a range of methods in which people from different backgrounds, such as subject matter experts, developers, product owners, agile coaches, or business analysts, **work together to develop domain models** instead of working on them in silos and then "throwing them over the fence" in the form of documents. The underlying idea is to make implicit knowledge visible and establish a common language. This allows misunderstandings to be identified and reduced at an early stage, risks to be minimized, and the foundation for sustainable systems to be laid.

The key principles are:

- **Together instead of divided:** Models are not created through handovers, but through direct collaboration. All participants, from domain experts to developers, work together to gain a consistent and clear understanding of the problem.
- **Visual rather than textual:** Simple visualizations on whiteboards, sticky notes, or digital boards make knowledge tangible and verifiable. This facilitates communication, makes complex relationships visible, and encourages active participation by all team members.
- **Exploratory instead of deterministic:** Modeling is an ongoing learning process, a conversation. Hypotheses can be tested and discarded. The model is not perfected in one single stage, but evolves in cycles. Through continuous feedback and adjustments, the model is gradually improved and adapted to new insights.

- **Focus on value creation:** Modeling is not an end in itself, but consistently focuses on solving a specific problem and creating measurable business value. It is not about documenting every detail, but about identifying the key elements for product goals, value propositions and user needs.

Collaborative modeling could be described as **moderated business analysis for agile environments**. Instead of writing down requirements in large documents, we create lively models in short iteration cycles that are endorsed and understood by everyone involved. This makes the methods relevant not only for software architects, but also for roles such as agile coaches, product owners, and requirements engineers.

In the following, we will look at selected methods in the context of the "Align" and "Discover" phases of the **DDD Starter Modeling Processes** mentioned in the previous chapter.

4.2 Align phase: Business Model Canvas and Value Proposition Canvas

The first phase of the DDD Starter Modeling Process is called **Align**. Before exploring a domain in depth, the overarching goals must be clear: *Why does our product exist? Which customers are we addressing? What value do we want to create?*

4.2.1 Business Model Canvas

The Business Model Canvas (BMC) is a strategic alignment tool that helps teams develop a **shared understanding of the overall business context** before diving into domain modeling. On a single page, it captures key aspects such as customer segments, value propositions, channels, revenue streams, and cost structures. In the context of Domain-Driven Design, the BMC does not describe the domain itself, nor does it provide direct input for identifying subdomains or Bounded Contexts. Instead, it establishes a common business framing: why an organization or a product exists, how it intends to create value, and which economic assumptions shape its decisions. This shared framing is valuable because it aligns stakeholders before more detailed, domain-centric exploration begins.

Facilitation tips:

- Work iteratively: start with a coarse, incomplete canvas and refine it over time.
- Focus on shared understanding rather than correctness or completeness.
- Explicitly mark assumptions, uncertainties, and open questions using colors or symbols.

From a DDD perspective, the Business Model Canvas serves as context-setting input, not as a modeling artifact. It helps teams agree on the broader business narrative, creating a stable starting point for subsequent domain exploration and collaborative modeling.

4.2.2 Value Proposition Canvas

The Value Proposition Canvas (VPC) zooms in on one specific aspect of the Business Model Canvas: the relationship between a customer segment and a value proposition. It explores questions such as: Which jobs do customers try to get done? Which pains do they experience? Which gains do they expect — and how does a product or service aim to address them? In contrast to domain-modeling techniques, the VPC does not describe domain behavior or internal business rules. Its value in a Domain-Driven Design context lies in sharpening the external view on value creation.

Facilitation tips:

- Start with customers and their needs, not with solution ideas.
- Use concrete examples to avoid abstraction.
- Encourage multiple perspectives — product, business, and technology often highlight different aspects of value From a DDD perspective, the VPC helps inform strategic prioritization, not modeling decisions. It supports discussions about where an organization differentiates through customer value and where commodity solutions may be sufficient. These insights later feed into strategic decisions such as identifying Core, Supporting, and Generic domains, which we will revisit in the Strategize section of the chapter on Strategic Domain-Driven Design.

4.3 Discover Phase: Big Picture EventStorming

Once the goals and value propositions have been clarified, the **Discover** phase is about understanding the domain in detail. One of the most popular methods for this is **Big Picture EventStorming**.

4.3.1 Description

EventStorming is based on the idea that technical expertise can best be described through **domain events**: things that happen in reality and are relevant to the business ("Customer placed an order"). These events are arranged chronologically and thematically as sticky notes on a long paper or digital whiteboard. This visual, collaborative approach quickly creates a shared picture of the domain.

4.3.2 Participants

Big Picture EventStorming is only useful if **all relevant perspectives** are represented:

- Domain experts who know the day-to-day business.
- Developers and architects who consider technical feasibility and integration.
- Product owners, business analysts, and agile coaches who ensure prioritization, value creation, and moderation.
- Representatives of adjacent departments or external partners if their systems or processes are directly involved.

The goal is to gain as complete a picture of the value streams as possible, thereby overcoming silo thinking.

4.3.3 Elements

Big Picture Event Storming uses a few central **element types** on sticky notes, which are distinguished by colors and symbols:

Element	Color	Description
Domain Events	orange	domain-relevant events that change the state
Commands	blue	actions that trigger an event
Actors / Roles	yellow	people or systems that execute commands
External Systems	pink	systems or organizations outside one's own sphere of influence
Hot Spots	red	UUncertainties, contradictions, or conflicts that need to be investigated

4.3.4 Phases of the process

A Big Picture Event Storming typically follows several phases:

1. **Chaotic Exploration** – All participants write down domain events on sticky notes. The focus is on quantity, not order. The goal is to bring implicit knowledge to the table.

2. **Enforcing the Timeline** – The events are placed in chronological order. This is where initial discussions about sequences, dependencies, and process logic take place.

3. **Adding Structure** – Pivotal events are marked, swimlanes are introduced, and external systems are added. The model gains structure and depth. **Pivotal events** play a special role in Big Picture Event Storming. They mark particularly important domain events in the timeline that fundamentally change the state of the domain.

4. **Identifying hot spots** – Open questions and conflicts are made visible. These points are particularly valuable because they mark risks or opportunities for innovation.

5. **Refinement & Exploration** – Depending on the objective, additional elements such as commands or user roles can be added. This develops the model into a basis for bounded contexts and architecture discussions.

4.3.5 Facilitation tips

As a facilitator, you should first ensure that you have a large physical or digital space because lack of space is the biggest enemy. Right at the start, it is important to explain that this is not about "right or wrong," but about exploration. Make sure that everyone involved actively contributes by specifically involving quieter roles. Colors help you to clearly distinguish between the different elements. Finally, it is important to keep the pace high: details can always be refined later; what is crucial is the shared momentum in the workshop.

4.3.6 Benefits

EventStorming is particularly suitable for the Discover phase because it **quickly adds depth** and **reveals implicit knowledge**. Teams recognize process breaks, contradictions, and potential for improvement. At the same time, it creates a basic foundation from which bounded contexts, ubiquitous language, and technical models can later be derived.

4.4 Discover Phase: Domain Storytelling

Once Big Picture EventStorming has revealed a broad picture of the domain, **Domain Storytelling** allows for a deeper dive into specific processes. The method focuses on **actors and their interactions** and reveals implicit ways of working through storytelling. This is particularly valuable for many teams, as stories are more accessible than abstract models. Anyone who witnesses a domain expert

describing a task step by step immediately gets a feel for how the work actually functions in reality.

4.4.1 Description

In Domain Storytelling, domain experts tell stories from their everyday work. Moderators draw these stories live using simple symbols: actors are represented as pictograms, their activities as arrows with short descriptions. The result is a visual narrative that shows how people and systems work together to complete specific tasks. The method is easy to understand because it draws on the natural way people pass on knowledge: through stories. Instead of reading dry process documentation, participants immediately see how a typical situation unfolds and which roles, work objects, and activities play a part in it.

4.4.2 Participants

Successful Domain Storytelling requires the right people in the room. **Domain Experts** who can contribute authentic stories from their practical experience are indispensable. In addition, **Developers and Architects** are needed to ask technical questions while listening, thereby revealing implicit assumptions. **Product owners, business analysts, and agile coaches** are also important participants because they can bridge the gap between the product vision and value creation and support the moderation. It is often sufficient to start with a small, focused group, as several storytelling sessions with different participants can easily be combined later. It is important that those in the room are the ones who really know what everyday work looks like.

4.4.3 Elements

The most important building blocks in domain storytelling are kept simple:

- **Actors** represent people or roles that take action.
- **Work objects** are things that are processed or moved in the process, such as an order or a document.
- **Activities** describe actions performed on work objects by the actors.

- Several activities result in a **story**, i.e., a sequence that depicts a scenario from the real world.

With these few elements, even complex processes can be presented in an easily understandable way.

4.4.4 Stages

A Domain Storytelling workshop usually begins with the domain experts **telling a story**, i.e., describing a typical process from their everyday work. Meanwhile, the facilitators **visualize** the actors, work objects, and activities on a whiteboard or in a digital tool. Once a first version is ready, a short **reflection** follows: The participants check whether the story has been reproduced correctly. Then **variants and special cases** are added so that exceptions and alternative processes also become visible. Over time, several stories emerge that together provide a comprehensive picture of the domain. This iterative structure makes it easy to start small and expand the model step by step.

4.4.5 Facilitation tips

As a facilitator, you should record the stories in the language of the domain experts without translating or abstracting terms. This is where ubiquitous language comes in. Use simple symbols and keep the pace high so that the flow of the narrative is not interrupted. Encourage participants to give concrete examples, as these make the story more tangible. Make sure that different perspectives are heard by specifically asking how other roles experience the process. Once several stories have been collected, it is worth highlighting similarities and differences to reveal patterns and recurring structures.

4.4.6 Benefits

Domain Storytelling is suitable for use in the discovery phase to make **specific work processes** transparent and build a common understanding. It helps to reveal implicit rules, roles, and dependencies. This also reveals misunderstandings

that often remain hidden in everyday life. For teams, the method provides an excellent foundation for formulating requirements more clearly, defining bounded contexts, and developing the ubiquitous language. Anyone who has experienced a session not only understands the theory, but also has a tangible picture of how the domain actually works, and that is precisely the crucial benefit of this method.

4.5 Conclusion

Collaborative modeling brings the principles described in the foundation, such as close collaboration, common language, continuous learning, to fruition in the real world. Whether through Business Model Canvas, Value Proposition Canvas, EventStorming, or Domain Storytelling, in all cases, a space is created in which business and technology think together.

For the target groups of this primer, from software developers and agile coaches to product owners and business analysts, collaborative modeling offers concrete tools for building bridges. Ultimately, it's not just about models, but about **alignment, comprehensibility, and sound decisions**.

5 Strategic Domain-Driven Design

Strategic Domain-Driven Design deals with the big picture: How can we structure complex domains in a way that creates viable technical systems and suitable organizational structures? While tactical DDD does the detailed work on the model, strategic DDD provides the context in which this detailed work becomes meaningful. The aim is to create clarity: Which parts of a domain are really of strategic importance? Where is it worth investing in quality and in-house development? And how do we prevent architecture or organization from missing the actual needs of the business?

5.1 Problem- and Solutionspace

An important foundation of strategic DDD is the separation of problem and solution space. In the problem space, we ask the question: *What actually needs to be achieved?* Here, we are dealing entirely with the needs of users, business models, and basic constraints. Technology is not important at this stage. Only in the solution space do we address the *how*: Which architectures, systems, and models implement the business purposes?

Domains and subdomains belong exclusively in the problem space. They describe what an organization does, where its expertise lies, and how it creates value. All other concepts of Domain-Driven Design, such as bounded contexts, aggregates, entities, value objects, context maps, and ubiquitous language, belong in the solution space. They describe how the purposes of the subdomains are implemented in technical and functional terms.

This distinction prevents hasty technology decisions and creates an early focus on the business domain. A practical example: Anyone who identifies a subdomain *Doctor's office management* is describing the purpose of structuring processes in a doctor's office and managing patient information. Whether this is later implemented as an app, web application, or part of a larger system is irrelevant at first.

Only when the business purpose is understood is it worthwhile to move into the solution space.

5.2 Domains and Subdomains

Domains are the percieved areas of activity and expertise of an organization. They are what a company "stands for": its claim on the market, combined with a clear purpose, specific expertise, and certain ways of working. A domain marks the area in which an organization has expertise and in which it has established certain processes and ways of working.

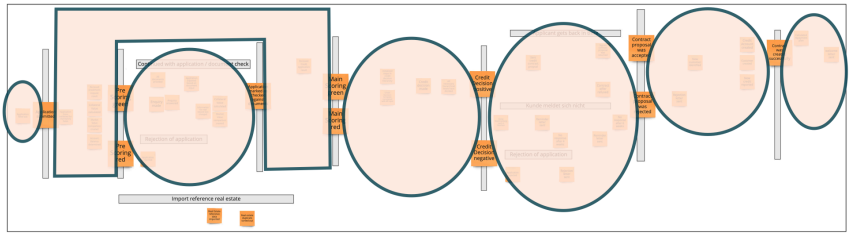
In large companies, several such domains often exist side by side. An automotive company, for example, operates in the domain of *vehicle development*, but also in the domain of *financial services*. Both domains differ fundamentally in terms of requirements, expertise, and working methods. The same applies, for example, to the regulatory environment. The "financial services" domain is regulated by a regulatory entity called BaFIN (in Germany), while the "vehicle development" domain is subject to the framework conditions of the German Road Traffic Regulations (StVO) and occupational safety.

Subdomains are intersected within a domain, each of which fulfills a clear purpose. Identifying and intersecting subdomains is a challenging task because it is not a matter of technical distinctions, but rather of identifying coherent areas of purpse. A subdomain should always focus on a precisely defined task. A rule of thumb is that if you cannot describe the purpose of a subdomain in a single sentence without using many "ands" or "ors," it is too broad and probably not functionally cohesive.

It is helpful to look at processes: **Pivotal events**, i.e., key moments such as *application submitted* or *document checked*, often mark transitions between subdomains.

Pivotal Events

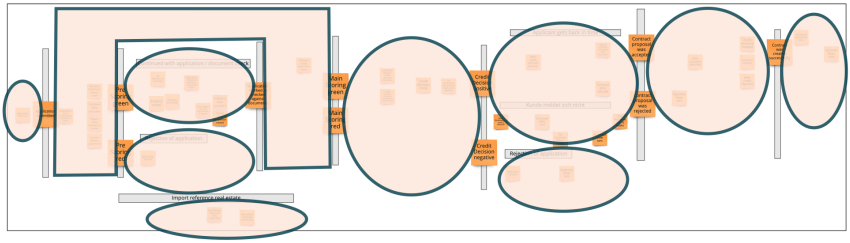
Heuristic: A pivotal event probably lies on the boundary of a subdomain



Swimlanes, which indicate different paths or branches in the process, also help with boundaries. They are often an indicator that something different is happening at these points.

Swimlanes

Heuristic: Swimlanes are indicators for different things happening and therefore may indicate different subdomains

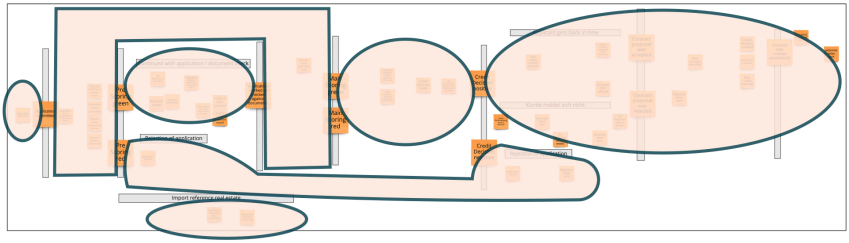


This distinction is important for the concept of cohesion. We want to achieve a high degree of functional cohesion in a subdomain. Therefore, you should always

take a very critical look at whether the functionalities that are to be bundled in a subdomain really have a very high degree of cohesion at the functional level.

Cohesion

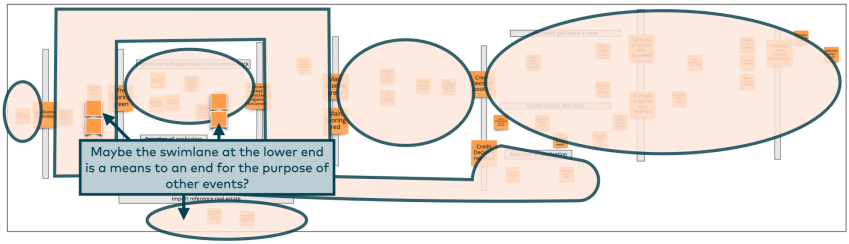
Heuristic: Domain events in a subdomain have a high degree of functional cohesion.



A means to check the degree of functional cohesion is to formulate the domain-specific purpose of a subdomain. The fewer "and," "or," and "as well as" appear in this text, the higher the degree of functional cohesion. A very valuable publication on this topic is the "Structured Design" paper by Stevens, Meyers, and Constantine. We will revisit the topic of "purpose" as it relates to bounded contexts.

Purpose

Heuristic: Each subdomain has a clearly defined functional purpose



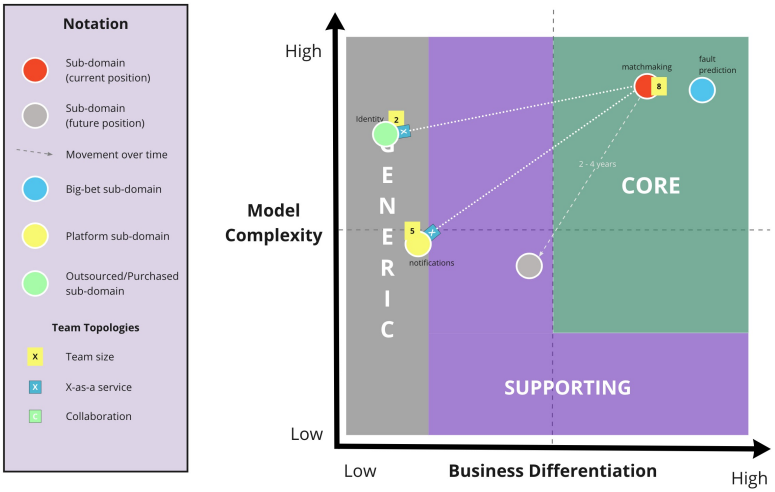
Subdomains correlate with each other towards a higher domain-specific purpose. Together, they represent the problem space, meaning that they are linked in terms of functionality without losing their autonomy. The goal is to keep these relationships lean and clear. Loose coupling does not only arise in the code, it already emerges at this level: the more coherent the subdomains are, the fewer dependencies they require between each other.

In the Domain-Driven Design Starter Modeling Process, the decomposition of a domain into subdomains happens in the “Decompose” phase.

5.3 Strategic Classification

Not all subdomains are equally significant. Strategic DDD distinguishes between core, supporting, and generic subdomains. Core domains are those areas in which an organization differentiates itself from the competition. These are of the highest strategic relevance, which is why you should not compromise on quality and ownership. Supporting subdomains are vital, but they do not differentiate. Here, compromises are possible, for example in the form of customization of custom of the shelf (COTS) software such as SAP for example. Finally, generic subdomains provide commodity functionality that is better covered by Software as a

Service (SaaS) offerings than by proprietary in-house development with your best engineers. This distinction is not an end in itself, but rather guides investment decisions. Core domains belong at the center of in-house development with your best engineering teams, while generic subdomains can often be realized more cost-effectively with off-the-shelf products. The classification of subdomains can change of time. A good example is a fashion company which I have worked with in the past that initially viewed its e-commerce as a commodity, later professionalized it, and finally developed it as a core domain based on insourcing was able to introduce new sales models within a few weeks and respond agilely to market changes.



(Image Source: https://github.com/ddd-crew/core-domain-charts)

In the Domain-Driven Design Starter Modeling Process, the strategic classification of subdomains happens in the “Strategize” phase.

5.4 Bounded Contexts

Subdomains get implemented through bounded contexts in the solution space. A bounded context defines a boundary within which a model exists in a consistent language and which is tailored to a specific purpose (that of the subdomain). This makes it the bridge between business and technology. The key principle here is **model specialization over generalization**.

Many organizations have had the painful experience of trying to enforce a universal model that covers all eventualities just for the sake of reusability. The result is an overloaded system without focus: for example, a "business partner" service that is supposed to represent end customers, repair shops, and therapists equally in an insurance company and thus includes hundreds of attributes. The result is a mostly data-driven model that no one can understand and that doesn't fulfill any task properly.

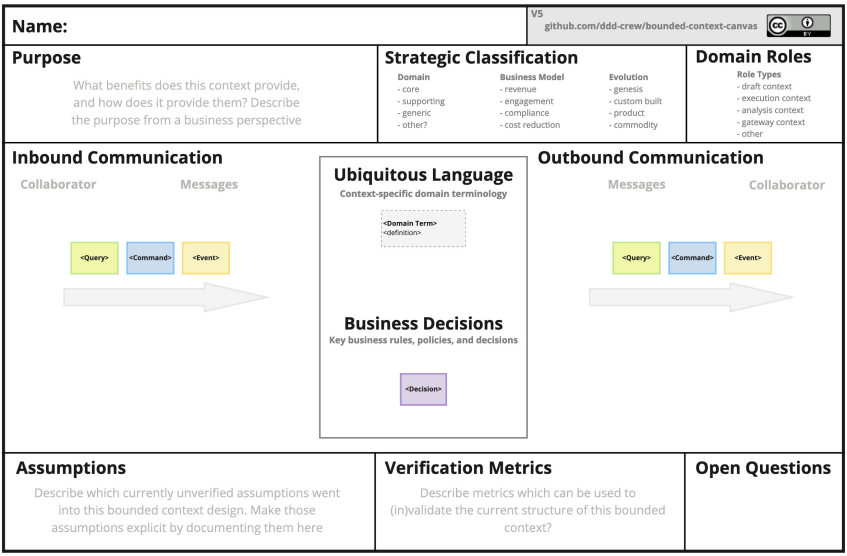
DDD recommends specialized models that are clearly tailored to their purpose instead. A participant at a conference is not a "customer" in the context of *check-in*, but receives a *badge*. In the context of *ticket sales*, the same person represents a financial transaction, and in the context of *catering*, an estimate of food requirements. Each model is specific to its purpose, and that is precisely what makes it valuable as well as evolvable.

Tools such as the **Bounded Context Design Canvas** support this transition from the problem space to the solution space. They help to capture the relevant terms, rules, and communication flows for each subdomain and to develop a consistent language. In practice, this often results in "living documentation" that provides guidance for both architects and domain experts.

5.5 Working with the Bounded Context Design Canvas

The **Bounded Context Design Canvas** is currently one of the more popular tools in strategic Domain-Driven Design. It helps to structure and clarify the transition from the problem space to the solution space. While many DDD workshops end with a wealth of insights about domains and subdomains, the canvas ensures that

this knowledge does not simply evaporate, but is transformed into a tangible, living artifact. Essentially, it serves to develop a clearly defined, commonly understood model for each identified subdomain: with its own language, its own business rules, and clear communication boundaries. Each canvas represents a possible or actual bounded context.



(Image Source: https://github.com/ddd-crew/bounded-context-canvas)

5.5.1 From the problem space to the canvas: the decompose phase

After the **decompose phase** of the DDD Starter Modeling Process, the focus is on working out the previously identified **subdomains** in detail. The fundamental cornerstones are transferred to the canvas: the **name**, the **purpose** (i.e., the functional purpose), and the initial ideas or potential candidates for the **ubiquitous language**: i.e., the terms used by the people involved to talk about this part of the domain. Please be aware that the ubiquitous language is more rigid than the terms we jot down at the canvas at this stage.

Even at this early stage, it is worth noting down the first **business rules** that we are aware of. These are an expression of the specific behavior expected in the subdomain and give structure to the emerging model. This is where the value of collaborative modeling becomes apparent: experts and developers sit together in front of the canvas and align their language. “customer” might become “participant”, “ticket“ might become “badge”, and suddenly there is clarity about meanings that were previously implicit and therefore potentially ambiguous.

A separate canvas is created for each subdomain. This separation is important because it ensures focus and coherence. A shared canvas for multiple subdomains almost always leads to blurred boundaries and models growing together in an uncontrolled manner. At this early stage, the focus is not on precision but on orientation. The goal is to create an artifact that stimulates discussion and serves as a communication anchor.

5.5.2 Making communication visible: the connect phase

Once you have identified subdomains and described their purposes, the **connect phase** follows. In this phase, you examine the relationships between the subdomains. The canvas helps you to systematically capture **inbound and outbound communication**.

Inbound communication describes which events, commands, or requests a subdomain receives from its environment. Outbound communication, on the other hand, shows which messages it emits on its own. We document this communication on the canvas to make the interaction between subdomains transparent.

Often, **domain events** become visible here, forming the backbone of loose coupling. Instead of one subdomain directly calling another, it publishes an event, such as “appointment confirmed” or “invoice created”, to which other subdomains can react. This way of thinking shifts the perspective from synchronous orchestration to asynchronous collaboration.

In this phase, the canvas serves not only as documentation, but also as a tool for reflection: Do the communication relationships match the intended purpose of the subdomain? Is its autonomy restricted by too many dependencies? When

such questions become apparent, the canvas is not a static form, but a mirror of shared understanding.

During the decomposition into subdomains we had a strong focus on high (functional) cohesion. The connect phase takes a look at the other important design principle for good boundaries: loose coupling.

In terms of tools you can use Domain Message Flow Modelling (https://github.com/ddd-crew/domain-message-flow-modelling), Activity Diagrams but also Domain Storytelling works astonishingly well here. Just use your subdomains as actors when you analyze communication with Domain Storytelling.

5.5.3 Taking strategic decisions: the strategize phase

The **strategize phase** is about strategically classifying the subdomains. The Bounded Context Design Canvas provides the **Strategic Classification** field for this purpose, which states whether a subdomain belongs to the **core**, **supporting**, or **generic domains**.

This classification is more than just a label. It influences how much energy, budget, and attention a subdomain receives in further development. Core subdomains, those that differentiate the organization, require intensive maintenance, high quality, and, in most cases, internal development expertise. Generic subdomains, on the other hand, should be deliberately simplified and, where possible, covered by out-of-the-box solutions.

By visibly recording this classification on the canvas, the discussion about priorities becomes tangible. The team can decide together where it is worth investing deeply in modeling and where "good enough" is really good enough.

5.5.4 Refining language and rules: a continuous process

Throughout all phases, we continue to work on the **ubiquitous language** and the **business rules**. These two areas are at the heart of every canvas. Language and rules are the foundation on which we establish understanding and model consistency.

New terms are added, old ones are questioned, and definitions are refined. At the same time, they help to clarify or complement functional rules. Over time, this creates a shared understanding that is viable for both the domain and the technical implementation.

This process is particularly valuable when different disciplines are involved: product management, development, UX, perhaps even support or sales. The canvas acts as a catalyst here: it forces everyone involved to make implicit assumptions explicit.

5.5.5 Working with uncertainty: open questions and hypotheses

No model is created in isolation. Often, questions remain unanswered or assumptions unconfirmed. The canvas has specific fields for this: **Open Questions**, **Assumptions**, and **Verification Metrics**.

Open Questions document ambiguities such as business, organizational, or technical uncertainties. *Assumptions* record hypotheses, such as "Scheduling is always linked to a doctor." These assumptions form the basis for targeted learning. Finally, *Verification Metrics* define how it can be verified later whether an assumption was correct. In this way, learning is systematically integrated into the modeling process instead of just happening on the side.

These three fields promote a reflective attitude in the DDD context: it is not about knowing everything from the outset, but about consciously dealing with uncertainty.

5.5.6 From the canvas to the implementation: refinement in the solution space

Once the work in the problem space reaches a stage of maturity, implementation begins in the solution space. This reveals another feature of the canvas: it is not a one-time result, but a living artifact. In the solution space, the contents of the canvas are continuously refined. Terms become more precise, rules clearer, and communication flows more accurately modeled. At the same time, it may happen

that a subdomain is divided into several **Bounded Contexts** in the course of the work, each with its own canvas. This split is not a mistake, but an expression of growing understanding. What initially appeared to be a single entity later turns out to be several more clearly defined contexts, each with its own language and responsibilities. It is precisely in this continuous process that the strength of the Bounded Context Design Canvas lies: it combines the exploratory openness of the problem space with the structured precision of the solution space. It is a thinking tool, a means of communication, and documentation all at once, and thus a central link in strategic Domain-Driven Design.

5.6 Socio-technical alignment

Strategic DDD does not end with bounded contexts, but extends into the organization. **Context maps** can be used to visualize relationships between teams and bounded contexts, which also map technical and organizational dependencies. This clearly shows that clear context boundaries not only make the architecture more stable, but also lay the foundation for autonomous teams.

Concepts from **Team Topologies** reinforce this approach: teams should be divided along bounded contexts so that ownership, autonomy, and minimal coordination are possible. Those who cleanly divide subdomains and contexts not only create a robust software architecture, but also lay the foundation for an organization that makes fast flow possible and provides excellent support for value streams.

5.6.1 Context Maps

Context maps are a tool for visualizing relationships between teams and their bounded contexts on a functional, technical, and organizational level. While bounded contexts describe the boundaries within the solution space, the context map shows how teams relate to each other, who depends on whom, and what kind of collaboration or coupling exists.

It thus makes implicit dynamics explicit: power relationships between teams, the propagation of models, and governance structures. Cleanly modeled code is of

little help if teams are caught in dependencies or models propagate uncontrollably throughout the landscape.

Team-Relationships in Context Maps

Context maps consider not only technical interfaces, but above all social and organizational relationships. Three basic types provide the foundation:

Upstream / Downstream: The upstream team designs the model and shapes the language. The downstream team consumes these results and depends on the stability and quality of the upstream team. As in a river system, control lies "upstream." Understanding this direction is central to any context map.

Free: Teams operate independently of each other. There is no direct coupling, either technically or organizationally. This freedom is valuable when collaboration is only loosely or temporarily required, but carries the risk of inconsistent models.

Mutually Dependent: Two teams are mutually dependent on each other. Decisions must be coordinated, and changes require synchronization. Such relationships are particularly sensitive because they can easily lead to coordination bottlenecks. They require explicit communication and release mechanisms.

These three relationship types determine how closely teams work together, how dependent they are on each other, and which integration patterns make sense. The context map serves here as a mirror of the real organizational dynamics, not as an idealized image.

Context Mapping Patterns

1. Open-Host Service (**OHS**) An upstream context provides a stable, documented interface for many consumers, a kind of a public API. Technically, this can be an API, an event stream, or a message format. It is neutral and extensible, ideal for many downstream systems.

2. Anticorruption Layer (**ACL**) The downstream system protects itself with a translation layer that converts the external model into an internal one. This layer

isolates legacy issues and enables evolutionary design. ACL is one of the most crucial decoupling patterns for integrations.

3. Conformist (**CF**) The downstream team adopts the upstream system model unchanged. This can happen out of necessity, convenience, or conviction. A conformist avoids translation logic but loses autonomy. In legacy system landscapes, this is often an indicator of tight coupling.

4. Shared Kernel (**SK**) Two contexts share part of the model or even artifacts (e.g., a library or database). This saves integration effort but creates high coupling. Acceptable if the shared part is small and stable; toxic if different teams (or suppliers) are pulling on it.

5. Partnership (**PNR**) Teams that have a shared kernel are mutually dependent and should form a partnership. They plan, develop, and integrate together. Success or failure is mutual. Partnerships require trust, transparency, and frequent coordinated releases. On the other hand they are a high indicator for cross-team coordination and thus for bottlenecks.

6. Customer/Supplier Development (**CUS-SUP**) The upstream team is the supplier, the downstream team is the customer. Requirements are agreed upon jointly, but there is a hierarchy. This pattern helps to shape influence in a targeted manner: Who is allowed to demand what, who delivers what and when? Deviations from this are indicative: "vetoing customers" or "overcautious suppliers" reveal governance and evolution risks.

7. Published Language (**PL**) A shared language in which systems exchange information. It is documented, supported by multiple parties, and decouples internal models from interfaces. In combination with an OHS, it often forms the standard for integration. Think about iCalendar or vCard as examples.

8. Separate Ways (**SW**) Sometimes the best integration is no integration at all. If the effort involved in linking exceeds the benefits, it is better to let teams work independently or to solve processes organizationally. "Separate ways" stands for conscious separation, often temporary, sometimes permanent.

9. Big Ball of Mud (**BBOM**) Refers to chaotic, unstructured systems without clear model boundaries. They are too large, too diffuse, and too risky for profound

interventions. It is important to isolate them so that their model does not spread further. Anticorruption layers are mandatory here, conformists a warning and shared kernels a no-go.

Benefits of Context Maps

Context maps are more than just architecture diagrams, they are tools for organizational diagnosis. They help to reveal three key perspectives:

Power and influence: Who controls models and interfaces? Where do dependencies arise?

Model propagation: Which models spread uncontrollably through copy-paste or shared kernels?

Governance: How are changes decided? Are there clear responsibilities or hidden veto structures?

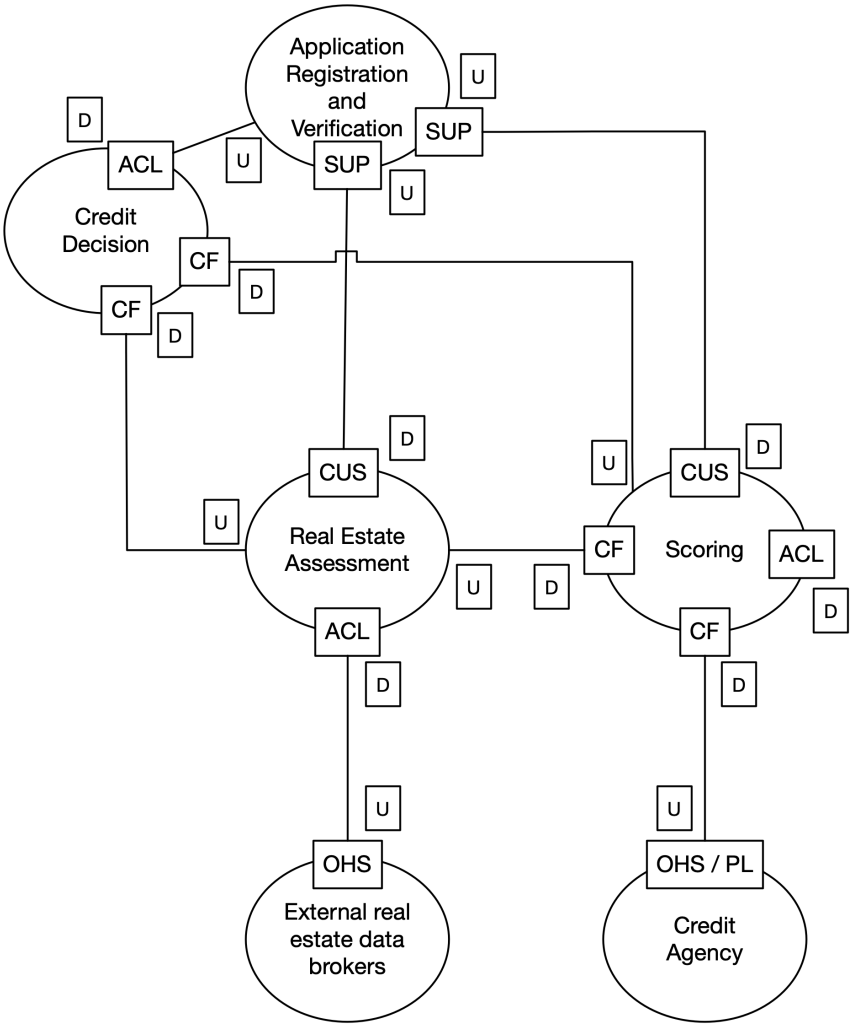
With these insights, teams can make informed decisions: Which relationships do we want to maintain? Where do we need to draw boundaries? Where does technical decoupling help and where does organizational decoupling help?

Practical application and visualization

Context maps can be modeled on whiteboards, in digital collaboration tools, or with sticky notes. Typically, teams and their bounded contexts are represented as circles, and relationship types are represented as lines with pattern labels (e.g., customer/supplier, ACL, partnership, SW).

An example: An upstream team offers a model via an Open Host Service (OHS) that a downstream team integrates via an Anticorruption Layer (ACL). Or two teams go “Separate Ways” – documented, accepted, stable.

This visual language also makes political and organizational flows visible which is the true value of context maps in strategic Domain-Driven Design.



(Image source: https://leanpub.com/ddd-by-example/)

Resources and Tools

An excellent starting point is the open-source project Context Mapping by the DDD Crew (https://github.com/ddd-crew/context-mapping). It offers a

lightweight notation, many visualization ideas, and a Miro Starter Kit that can be used to model your own maps directly. The examples available there show how technical and organizational relationships can be represented equally.

For teams that want to maintain context maps automatically or based on code, the Context Mapper tool (https://contextmapper.org/) is also a good option. It allows you to describe bounded contexts and their relationships in a domain-specific language and generate diagrams or architecture documentation from them. This creates a bridge between conceptual design and documented architecture.

5.6.2 The Relationship between Domain-Driven Design and Team Topologies

Domain-Driven Design and Team Topologies complement each other seamlessly. Both approaches address the question of how complex systems can be designed and further developed. DDD does this from a domain-oriented perspective, while Team Topologies approaches it from an organizational perspective. DDD helps to identify domain boundaries in the software model, Team Topologies indicates how these boundaries should be incorporated into the team structure and communication channels of an organization.

Joint goal: Socio-technical alignment

Both DDD and Team Topologies strive for socio-technical alignment. This means a close link between the domain structure, technical architecture, and organizational setup. In a well-aligned system, team boundaries reflect the bounded contexts defined in strategic DDD. The goal: Teams have end-to-end responsibility for clearly defined functional areas and can deliver value independently of each other.

DDD describes what should be separated: the functional domain boundaries. Team Topologies on the other hand describes how this separation can be implemented in terms of organization and communication. Together, they make it possible to combine architectural clarity and organizational autonomy.

Team Types according to Team Topologies

In an ideal setup, each team is assigned to a bounded context. This context defines the domain model, language, and responsibilities of the team. This creates autonomous units that can act quickly and independently within their context without having to rely on central coordination. The four team types from Team Topologies, which are stream-aligned, enabling, complicated subsystem, and platform, can be mapped to DDD concepts:

Stream-aligned teams work along a value stream that usually corresponds to one or more bounded contexts. They have full responsibility for the features that arise from this domain.

Enabling teams help stream-aligned teams adopt new practices, such as collaborative modeling or event storming. They promote DDD adoption in the organization and can also provide valuable support in day-to-day architecture and development work.

Complicated subsystem teams take care of sophisticated components that are not central to the domain but require a high level of expertise. They often work on supporting or generic subdomains.

Platform teams provide shared infrastructure and services that make other teams' work easier without restricting their autonomy.

This assignment ensures that the boundaries derived from DDD do not remain theoretical, but become effective within the organization.

Communication Flows and Team Interaction Modes

Team Topologies describes three basic ways in which teams work together using Team Interaction Modes. These modes help to consciously design communication channels and control dependencies in a deliberate manner, rather than by accident.

Collaboration refers to intensive, time-limited cooperation between teams that serves a clear purpose. It takes place when a problem is new or complex and re-

quires multiple perspectives, such as when introducing a new API or developing a new domain model. Collaboration is focused on learning and joint exploration.

X-as-a-Service stands for a stable, service-oriented relationship: a team offers a clearly defined service or platform function that other teams can consume. This form of interaction reduces cognitive load because consumer teams can focus on their own domain while the service provider delivers stability and reliability.

Facilitating describes a supportive collaboration in which one team (usually an enabling team) helps other teams adopt new skills, practices, or technologies. The focus is not on the result, but on empowerment. The goal is to strengthen the supported team and then release it back into autonomy.

These three interaction modes form the basis for a deliberate communication strategy between teams. They enable teams to collaborate to varying degrees depending on the situation: temporarily, permanently, or in an advisory capacity. Only in combination with the principles of DDD does this result in a complete view: The patterns of the context map describe the domain-specific and model coupling, while the interaction modes map the social and organizational level.

Fracture Planes

A key connecting element between Domain-Driven Design and Team Topologies are the so-called fracture planes. They describe possible dividing lines within a software or organizational architecture, along which teams and systems can be sensibly separated from each other. While DDD identifies these lines from a domain perspective, Team Topologies provides a language for organizational implementation.

Fracture planes can run along various dimensions: functional (e.g., subdomains), frequency of change, regulatory requirements, technology, or even organizational responsibilities. Combining them with bounded contexts is particularly valuable, as they provide a natural fracture plane that teams can use for orientation.

Where bounded contexts arise, team boundaries should also run. This creates teams that act autonomously within their fracture plane and understand their

responsibilities without being dependent on other parts of the system. This strengthens the alignment between domain structure and organization.

Evolutionary domains and organizations

DDD and Team Topologies also share the insight that boundaries are not static. Functional domains change, organizations grow, and new technical possibilities arise. Accordingly, bounded contexts and team structures must also be regularly reviewed and adapted. What is a core domain today may become supporting or generic tomorrow, and vice versa.

The key lies in recognizing changes early on and actively shaping them. Team Topologies offers tools such as Team API and Evolutionary Change for this purpose, while DDD uses Context Maps and Strategic Classification to show where adjustments make sense from a technical perspective. Together, they enable an organization to remain adaptable without losing its coherence.

The socio-technical aspect is addressed in the DDD Starter Modeling Process in the “Organize” phase.

5.7 Conclusion

Strategic DDD is more than a set of methods, it is a framework for dealing with complexity at a large scale. It forces us to first clearly understand the domain purposes in the problem space before making technical decisions in the solution space. Domains are the areas of activity and expertise of an organization: what it stands for, where it has developed specialized knowledge and characteristic workflows. Within these domains, subdomains are sliced that have high cohesion and whose correlations are lean. Bounded contexts in the solution space provide specialized models instead of drifting into generic data dumps. Classification into core, supporting, and generic subdomains helps to steer investments in a targeted manner. Finally, socio-technical alignment connects the functional and technical structure with the organization itself.

This makes strategic DDD a bridge between business, technology, and organization. This creates the foundation for systems that not only work today but will also be viable tomorrow.

6 Tactical Domain-Driven Design

Strategic Domain-Driven Design describes how an organization can comprehend, segment, and structure its business domains. It provides language, instruments, and paradigms to establish orientation. However, this map alone is insufficient. The actual craft begins at the tactical level: the concrete design of what occurs within a bounded context. Here, expertise is no longer analyzed, but constructed.

Tactical Domain-Driven Design is the art of forming living structures from concepts: weaving architecture, models, and language together in such a way that they form a stable whole.

Those who take DDD seriously recognize that strategic insights only take effect when they are translated into concrete technical building blocks. Boundaries between contexts only make sense if what lies within those boundaries is clear, understandable, and consistent. This inner clarity is the task of tactical design. It is the level at which teams decide which objects bear responsibility, how consistency is maintained, which rules apply permanently, and how systems are allowed to change without falling apart.

Tactical DDD is therefore not a subordinate detail, but the practical form of the ubiquitous language. It translates knowledge into structure, language into code, and code back into new insights. The real value arises when this feedback is consciously designed: experts and developers understand the same terms, see the same connections, and correct any misunderstandings.

Where strategic DDD draws maps, tactical DDD describes the architecture. It works in fine layers: entities, values, aggregates, events, services, and factories form the building blocks with which business domain knowledge can be expressed precisely. Added to this are architectural patterns that focus on the domain, such as hexagonal architecture or event sourcing.

Tactical Domain-Driven Design does not view software as a static entity, but as a system in motion. A system that has learned to adapt. At its core, it is not about abstraction, but about precision: the ability to model meaning so clearly that it remains recognizable in code.

6.1 Tactical Patterns

At the heart of tactical DDD are a series of proven building blocks called tactical patterns. Together, they provide a language and concepts for expressing and implementing domain logic. These patterns should not be viewed in isolation, but rather as elements of a coherent vocabulary: entities, value objects, aggregates, domain events, repositories, services, and factories.

They all have the same goal: **coherence in modeling**. They give the software a structure that respects the domain model while embedding it in the technical framework of the architecture.

6.1.1 Entity: Identity and lifecycle

Within a domain, there are things that persist over time. Their state may change, but their identity does not. This temporal continuity is the defining characteristic of an entity. An entity represents a concrete domain object that passes through different phases during its existence while remaining recognizably the same. It has a name and meaning in the ubiquitous language and is defined not by its data, but by its identity.

In classic object-oriented thinking, an entity was often reduced to an object with an ID. In the context of Domain-Driven Design, this view is too narrow. An entity stands for a specific domain individual with its own history. Its behavior is closely tied to its lifecycle: which operations make sense or are even possible depends on where the entity currently is in that lifecycle. Methods therefore express domain intentions, not mere state changes.

The lifecycle of an entity is not a technical detail but a core domain concept. Entities come into being, evolve, and eventually reach an end—explicitly or implicitly. These transitions are part of the model and must be expressible in the language of the domain. An entity "knows" which stage it is in and which next steps are valid from a business perspective.

Designing an entity therefore always starts with language and time. How do domain experts talk about this thing as it exists and changes? How do they recognize it as "the same" despite many changes? Which states do they deliberately

distinguish? Entities are not just classes, but domain concepts with identity and duration—and it is precisely these two aspects that make them one of the central building blocks in tactical Domain-Driven Design.

6.1.2 Value Object - Meaning and Expressions

Value objects represent concepts where value, rather than identity, is decisive. They are the precise way to express functionality at the finest level. A value object exists only through its properties and the behavior that results from them.

The key property is immutability. A value object does not change; it is replaced. This seemingly technical principle is a profound functional proposition: meaning does not change retroactively. When a price, quantity, or time period changes, a new value is created, not a modified old one. This preserves the consistency of the domain over time.

Value objects are equal if their properties are equal, not because they are the same object. They are compared based on their meaning. This makes them reliable building blocks for calculations, validations, and communication between larger-scale concepts such as aggregates which we will discuss next.

Furthermore, value objects are carriers of language. A system that speaks in terms of the domain is more sustainable and readable than one that relies on primitive types. An operation such as amount.add(otherAmount) is not only technically but also semantically understandable. The syntax reflects the language of the domain.

Value objects are thus the opposite of meaninglessness. They are precise terms that preserve meaning and prevent errors at the semantic level. They make the code expressive and robust at the same time.

6.1.3 Aggregates – Consistency boundaries and invariants

Aggregates group entities and value objects into units of domain-specific consistency. They define where transactions end and where consistency must be

ensured. An aggregate root is the only entry point through which the internal state may be changed.

Designing aggregates is an exercise in balance. If they are too large, you lose flexibility; if they are too small, you lose meaning. The intersection along business invariants is crucial. These invariants are the rules that must always apply, not sometimes, not possibly. If two pieces of data or behaviors must be valid together, they belong in the same aggregate. If they vary independently, they do not.

In practice, aggregates often represent the most difficult but also the most important design decisions in tactical DDD. The art lies in finding consistency boundaries that are functionally necessary but technically viable. Aggregates are not purely structural patterns, but rather an expression of functional responsibility. Their task is to maintain integrity while enabling autonomy.

The central criterion here is the invariant. An invariant describes what must not be violated under any circumstances. In a banking system, for example, the following could apply: "An account may never have a negative balance unless it has an approved overdraft." This sentence contains everything you need to slice an aggregate: the account, the balance, the rule. These three elements form a semantic unit that must remain consistent in a transaction context.

Aggregates are therefore not a collection of entities, but representations of business coherence. The aggregate root (entity) is the guardian of this coherence. It decides which operations are allowed and mediates between internal state and external behavior.

As the system grows in size, the relationship between aggregates becomes more complex. Communication across boundaries then usually takes place via domain events, which we will discuss next. In such cases, the strict consistency of a monolith is exchanged for a flexible form of integrity. This means that changes are not visible everywhere at the same time, but spread via events. This principle of eventual consistency is not a shortcoming, but a conscious architectural decision. It allows systems to be distributed and yet reliable.

Sara Pellegrini's concept of **Dynamic Consistency Boundaries** (**DCB**) complements this view. It recognizes that consistency is not an absolute value, but

a contextual one. A system can decide situationally how much consistency it needs. This creates a model that deals flexibly with load, availability, and business significance. Instead of enforcing a single, universal pattern, DCB allows for a variety of consistency strategies: from strict isolation to soft, time-delayed synchronization.

Essentially, this means that aggregates are not boundaries of the software, but boundaries of understanding. They define what a team perceives as inseparable. And because this understanding changes with the domain, the design of aggregates is also evolutionary. In a living system, aggregates are recut, invariants are shifted, and events are grouped differently. Tactical DDD sees this not as a weakness, but as an expression of maturity.

6.1.4 Domain Event - The Storyline of the Domain

Domain events are the most basic form of communication in a DDD system. They are the language used to express domain expertise over time. A domain event is not a notification or a signal. It is a statement about a fact that has occurred in the domain. This wording is crucial: an event describes something that has happened, not something that is supposed to happen.

A domain event has three essential characteristics. It is past-oriented because it only refers to events that have already occurred. It is immutable because a fact cannot be undone. And it is meaningful because it is formulated in the language of the domain.

Domain events make time into a first class concept in the system. Instead of fixing states in tables or objects, you tell a story made up of events. Each new fact is added to this chronicle. This enables traceability, auditing, reproduction, and perhaps most importantly understanding.

Events are also the means by which aggregates, bounded contexts, and entire systems are loosely coupled. When an event occurs, other components can respond to it without knowing its originator. This creates independence and enables evolution. In this form, domain events become the infrastructure of knowledge: they connect the past, present, and future of technical expertise.

In event-driven architectures, domain events form the basis for asynchronous communication. They are both technical (as messages, for example) and semantic (as concepts). However, their greatest strength lies not in integration, but in clarity. Every domain event forces meaning to be made explicit. You cannot publish something you do not understand.

6.1.5 Repository – The abstraction of data access

A repository is the bridge between the domain model and persistence. Its purpose is to abstract the storage and loading of aggregates in such a way that the business logic remains unaffected. The business-logic code thinks in terms of aggregates, not tables.

A good repository behaves as if it had a collection of objects in memory. Methods such as findById, save, or delete are formulated in domain terms and follow the ubiquitous language. The implementation – whether via a relational database, a document store, or an event log – remains interchangeable.

This makes the repository a model of separation of responsibilities. It allows the domain to deal exclusively with behavior, while persistence remains a technical detail. In testable architectures, the repository is typically abstracted by interfaces that are defined in the domain or application layer and implemented in the infrastructure.

An important aspect is that repositories always work with aggregates, never with sub-objects. They guarantee the consistency of the transaction and prevent internal structures from leaking to the outside.

6.1.6 Service - Orchestration

Not all behavior in a domain naturally belongs to a single entity or aggregate. Some operations inherently span multiple aggregates, while others express domain logic that is conceptual rather than owned. Services exist to model such behavior.

From a plain Domain-Driven Design perspective, a service is a domain concept expressed as an operation. It orchestrates collaboration between aggregates, entities, repositories, and factories without becoming an owner of domain state itself. A service represents doing rather than being: it captures meaningful domain actions that cannot be cleanly located on one aggregate root without distorting the model.

DDD itself does not mandate a strict categorization of services. The often-quoted distinction between application services and domain services originates primarily from architectural styles such as Hexagonal Architecture. In that context, application services are used to coordinate use cases and technical concerns, while domain services remain part of the pure model. DDD, however, is more pragmatic: it focuses on expressing domain behavior clearly, regardless of whether that behavior is triggered by a use case or composed across aggregates. We will adress Hexagonal Architecture later in this chapter.

What matters is responsibility. A service should contain domain logic only when that logic is genuinely shared or coordinating in nature. It should not degenerate into a procedural script nor replace behavior that properly belongs inside aggregates. Services do not "own" invariants; they respect and invoke them. They rely on aggregates to protect consistency and on repositories to access them, but they do not undermine those boundaries.

Used well, services make structure explicit. They highlight places where collaboration is required and where no single model element should dominate. Their presence often signals seams in the model: points where multiple responsibilities meet without being merged. In this sense, services are less about filling gaps and more about preserving clarity in a domain model that remains cohesive, expressive, and evolvable.

6.1.7 Factory – Externalisation of construction logic

Factories are mechanisms used to correctly construct new aggregates or entities. They summarise construction logic, check prerequisites and guarantee that an object starts in a valid state.

A factory is not simply a construction tool, but an instrument of integrity. It protects against faulty states and ensures that all invariants are checked during creation. Especially in the case of complex aggregates or event sourcing, where a series of events must be initially generated, the factory becomes the formal start condition of the system.

Its existence frees the model from trivialities. If objects do not need to know how they are created, they can concentrate entirely on their behaviour. Factories thus support a principle that runs through the entire DDD: placing every responsibility where it belongs semantically.

6.2 Architectural Patterns

Architecture in the sense of tactical DDD is not a technical end in itself. It is the visible result of an attitude towards the business domain expertise. Every architectural decision is ultimately a statement about how much one understands the domain and how consistently one is prepared to translate this understanding into code.

Tactical Domain-Driven Design views architecture as a linguistic phenomenon. Structures do not arise from technology, but from semantics. When domain expertise is the source, architecture becomes a form of meaning.

In this sense, architecture is not rigid. It is a conversation about stability and change. The strength of DDD lies in the fact that it does not prescribe an architecture, but a criterion: the chosen structure must support the flow of expertise.

6.2.1 Layered Architecture

The classic layer architecture is something many systems have in common. It splits software into layers like presentation, application, domain, and infrastructure. Each layer only knows about the one right below it. Data flows down, control flows up. This setup has one clear advantage: it's easy to understand.

For teams starting with DDD, this model often provides a solid foundation. It creates discipline, forces the separation of (technical) concerns, and facilitates testing in clearly defined areas.

But it is precisely this simplicity that can also become a limitation. As the domain grows, subtle dependencies arise between the layers. Business logic seeps into the application layer, technical details creep into the domain. The boundaries blur until layers exist only as project folders.

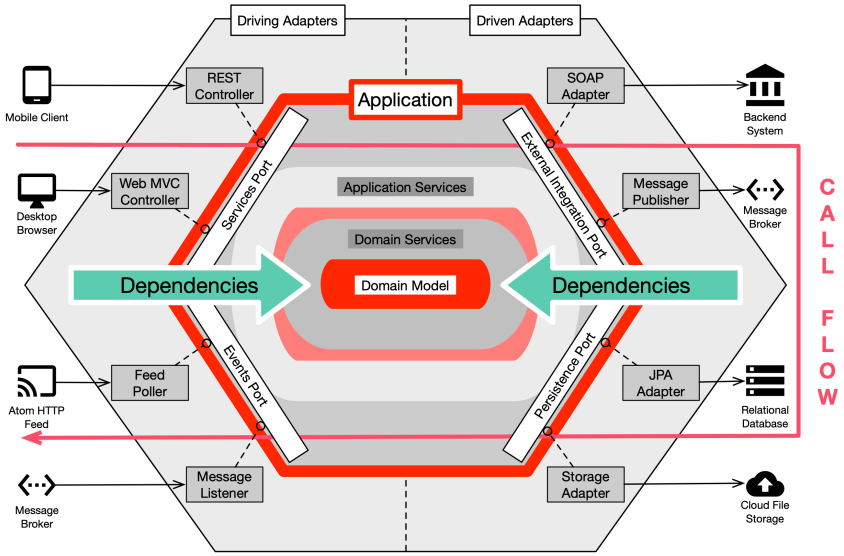
The problem lies less in the architecture than in its implicit hierarchy. The layered architecture places technology at the base and business logic in the middle. This creates a subtle dependency of language on infrastructure.

What initially appears to be organizational order is in fact a semantic distortion: the domain is not the foundation, but the center. When the business logic changes, everything around it must be able to move. In classic layered architecture, this is rarely the case.

Nevertheless, it remains useful in certain contexts. In systems that are manageable, stable, or highly data-centric, a layered architecture can be perfectly adequate. Its weakness only becomes apparent when systems begin to come to life, when models evolve, and teams are expected to act independently. Then the layered model becomes too narrow for what DDD aims to achieve: domain-driven evolution.

6.2.2 Hexagonal Architecture

Hexagonal architecture, made famous by Alistair Cockburn as Ports and Adapters, arose from the desire to overcome this limitation. It reverses the relationship between business logic and technology: all dependencies must point inward.



The domain model is at the center of the hexagon. It is the core of understanding, pure domain expertise, unaffected by frameworks and database access. The domain model usually contains entities, value objects, aggregates, and internal domain events. In the domain layer, the domain model is usually supplemented by domain services.

The application layer lies around it. It orchestrates the use cases. It contains application services and ports, coordinates processes, executes transactions, and manages policies. The application layer is the layer of action. It connects the meaning of the domain with what the system does.

The adapters are located at the periphery. They implement the ports through which the system communicates with the outside world. An adapter can be a user interface, a REST endpoint, a connection to a message broker, or a database access object.

This structure makes a simple but profound statement: all dependencies point inward. All knowledge about technology, frameworks, or infrastructure remains outside the core.

The hexagon is therefore not a purely technical form, but a semantic promise. It protects meaning from erosion by technology. Systems built in this way can change over the long term because their essence remains stable. Business logic can be tested without starting infrastructure; infrastructure can be replaced without affecting the domain.

But here, too, the hexagonal architecture is a tool, not a dogma. Not every system needs this level of separation. In small projects, it can seem too complex, and in highly exploratory phases, it can even be a blocker for flow. The key thing is that the team understands why it is deciding for or against this type of architecture.

Architecture should match the maturity level of the organization. The hexagon shows a direction. It forces language, clarity, and conscious boundaries. But it also requires discipline, and discipline only comes when you feel the benefits.

6.2.3 Event Sourcing

Event sourcing takes a different approach to persistence than most traditional systems. Instead of storing only the current state, it stores every relevant change as a sequence of events. The current state is derived by replaying these events in order. In other words, state is not written directly; it is reconstructed from what has happened.

Each event represents a fact that occurred in the domain: an application was submitted, a score was calculated, a contract was approved. These events are stored permanently and in the language of the business. Because of this, the system does not just know what the current state is, but also how it came to be.

This approach has practical consequences. Audit trails come for free, because the full history is available. Errors can be analyzed by inspecting past events, and alternative scenarios can be explored by replaying the same events with different logic. Debugging often becomes easier, because behavior can be traced step by step instead of inferred from snapshots.

Event sourcing also raises the bar for modeling. Every state change must be expressed explicitly as a domain event. Vague updates are no longer sufficient; the model must name what actually happened. This often leads to clearer terminology

and better alignment with how the business talks about its processes. Events become durable records of business decisions, not just technical notifications.

At the same time, event sourcing introduces additional complexity. Event schemas must evolve carefully, because old events remain part of the system forever. The separation between command handling and querying becomes important, as rebuilding state from events has different performance characteristics than reading a current snapshot. Teams must also accept that history is immutable: past events cannot be changed, only compensated by new ones.

Used thoughtfully, event sourcing works well with flexible consistency boundaries. Aggregates enforce consistency at the time commands are handled, while the event stream preserves a complete and reliable history. This leads to systems that are resilient not only in terms of data, but also in terms of change. When time is treated as a first-class concept in the model, evolution becomes a design concern rather than an afterthought.

6.2.4 CQRS

Command Query Responsibility Segregation (CQRS) builds naturally on the ideas behind event sourcing, but it is also useful on its own. At its core, CQRS recognizes that changing state and reading information are fundamentally different concerns. They serve different purposes and place different demands on the model.

On the command side, the system handles change. Commands express an intention to do something. They are validated against business rules, checked for invariants, and—if accepted—lead to state changes. In an event-sourced system, these changes are captured as domain events. In a state-based system, they result in updated state. In both cases, the command side is about correctness, consistency, and protecting the integrity of the domain.

The query side has a different focus. It exists to answer questions. It does not enforce invariants or coordinate transactions; it is optimized for fast access, clear structure, and expressive views of the data. Read models are shaped by the needs of consumers: user interfaces, reports, dashboards, or external systems. They

often denormalize or precompute information to make queries simple and efficient.

The strength of CQRS lies less in the mechanics and more in the mental model it introduces. It forces a clear distinction between doing and knowing. Writing answers the question "What is allowed to happen?" Reading answers "What do we currently know?" By separating these concerns, the model becomes easier to reason about, especially as complexity grows.

In combination with event sourcing, this separation becomes very explicit. Events form the bridge between the two sides. They record what happened on the write side and serve as the input for projections on the read side. Different projections can coexist, each representing a valid perspective on the same underlying history. This makes it possible to evolve read models independently, without risking the core business rules.

CQRS does not require event sourcing. It can also be applied in systems that store only current state. In those cases, commands still modify one model, while queries read from another. What matters is not the storage technology, but the decision to separate responsibilities deliberately.

Seen this way, CQRS is not a pattern you "install" in a system. It is a design stance. It aligns closely with Domain-Driven Design by encouraging clear boundaries, precise language, and models that reflect intent. Things that mean different things are modeled separately—and only connected where shared meaning truly exists.

6.3 Deployment Options

DDD predates the term "microservice." When Eric Evans published his book in 2003, nobody was thinking in terms of microservices, but rather in terms of contexts. Domain-Driven Design emerged long before microservices, containers, or cloud environments. It never relied on distributed systems to be effective. On the contrary, early DDD models lived in modular, monolithic applications which aimed to be well-structured, linguistically clean, and technically coherent. DDD is a framework for thinking, not an infrastructure concept. The task is not to

divide systems, but to design boundaries. How these boundaries are deployed is a secondary question, not the primary one.

Therefore: **DDD does not need microservices.**

In practice, a simple heuristic applies: model first, deployment later. Those who distribute too early only distribute misunderstandings.

6.3.1 Monolith

The word "monolith" has an undeserved negative connotation. It suggests rigidity, impenetrability, and technical debt. But in reality, a monolith is nothing more than a system that runs in a process space. Whether this process is well-ordered or chaotic is determined not by the deployment form, but by the design.

A well-structured monolith can be one of the clearest expressions of Domain-Driven Design.

If the internal modules are structured along functional boundaries, if communication between them takes place via explicit interfaces, and if each module is understood as a bounded context, then the result is what is now known as a modulith.

The modulith consists of several contexts that live within a deployment but are clearly separated from each other. Each context can be tested, developed, and modified independently without disturbing the rest. The deployment is shared, but the architecture is decentralized. The result is a system that combines the simplicity of shared processes with the clarity of separate models.

Such a structure is often the best choice for organizations that operate in stable business models or are in a learning phase. It allows for rapid refactoring, easy deployment, and shared data storage without sacrificing the principles of DDD. Above all, it provides a platform for evolution: as expertise grows, individual modules can become independent—in their own deployments, their own teams, their own life cycles.

The monolith is then not a contradiction to microservices, but their logical origin.

6.3.2 Microservices - Distributed Boundaries

The idea of microservices has changed software architecture in recent years. It promises independence, speed, and team autonomy. But it also brings with it new forms of complexity. From a DDD perspective, microservices are nothing more than physically separated bounded contexts. They make visible what DDD has long described: that boundaries must not only be conceived, but also lived.

When each team is responsible for a context, true ownership emerges. The language within the service can develop independently. Decisions that used to be made centrally are now made where the knowledge is. This autonomy is the greatest gain – but also the greatest challenge. Because distributed systems are never just technical artifacts; they are organization cast in code.

Where Modulith uses internal interfaces, microservices communicate via networks. This makes errors visible, but also expensive. Data must be separated, consistency becomes loose, and integrations must be orchestrated. The price of independence is increased complexity. DDD helps here because it understands this separation not as a technical requirement, but as a business necessity. A microservice should never be created because you want microservices, but because a context has very specific quality requirements. This is not a DDD decision but an architectural one.

In the best case, the transition is organic: microservices grow out of a modular system after boundaries have become stable. When teams know their contexts, when communication is understood, and when models can evolve independently of each other, then physical separation can take place without compromising semantic integrity.

This is the real meaning of the “model first, deploy later” principle. Only when you have mastered the boundary is it worth distributing it.

6.3.3 Self-contained Systems – Thinking a little bit bigger

Between a single monolithic system and a landscape of fine-grained microservices lies another architectural option: self-contained systems (SCS). An SCS is a system that delivers a complete piece of functionality end to end. It includes

everything it needs—from user interface to domain logic to persistence—and interacts with other systems only through well-defined interfaces.

The core idea is simple: instead of splitting a system by technical layers or infrastructure concerns, it is split by business responsibility. Each self-contained system owns its functionality, its data, and its internal model. Other systems do not reach into its database or reuse its internal code; they communicate only through explicit integration points such as APIs or events.

This makes SCS closely aligned with Domain-Driven Design, but with a deliberately pragmatic focus. The goal is not maximal decomposition, but clear ownership and autonomy. A self-contained system is independently deployable and independently evolvable, while still being part of a larger system landscape.

Compared to microservices, self-contained systems are usually larger in scope. They often cover an entire functional area rather than a narrowly defined capability. This makes them easier to understand and operate, especially in organizations where teams are already structured around broader business responsibilities. An SCS is less like a single service and more like a small application that plays a well-defined role in a network of systems.

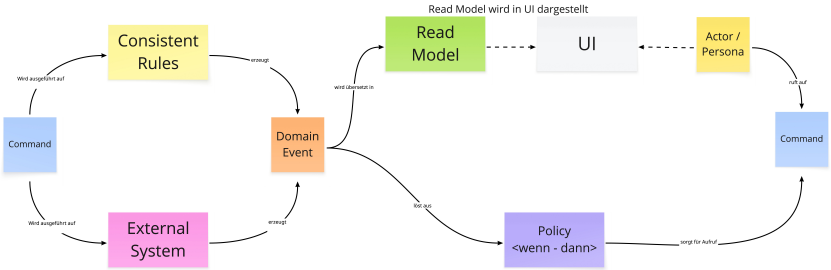
In practice, self-contained systems often emerge naturally. A module that has a clear purpose, stable boundaries, and little coupling to the rest of the system can be separated without major redesign. When this happens, the architecture evolves step by step. Existing models and responsibilities remain intact; only the deployment and integration boundaries change.

This evolutionary path is one of the main strengths of SCS. The architecture grows by extracting proven units, not by enforcing a target structure upfront. Boundaries are shaped by domain understanding and organizational reality, not by technology trends. As a result, systems remain understandable, adaptable, and connected—without the overhead of unnecessary fragmentation.

6.4 Design-Level EventStorming - From the model to the implementation

Design-Level EventStorming bridges the gap between strategic thinking and technical implementation. It translates the insights gained from Big Picture EventStorming and context delimitation into a concrete, implementable model within a bounded context.

Metaphorically speaking, the big picture draws the map and the design level drafts the blueprint for a building. Its goal is to understand the behavior of a system in such a way that it can be directly translated into code and in such a way that this code speaks the language of the domain.



6.4.1 Preparation – The scope and the stage

A design level workshop always begins with a clear focus. In contrast to the big picture, which examines an entire domain, here the focus is on a limited flow of events – a sequence of decisions and reactions within a bounded context. This scope can be a single use case, a central event, or a sub-process that is technically significant.

The modeling space in which the work is done physical or digital is a central element. All participants must be able to see the same area and interact with it at the same time.

The moderator guides the team in small steps from the first event to the complete event chain. The motto is: "Let the domain speak, not the technology." Each event is formulated in the past tense, each command as an action that could lead to an event. This gradually creates a visible flow of cause and effect.

6.4.2 Select a starting point

Every Design-Level EventStorming begins with a conscious delimitation. The scope should be small enough to go into depth within a workshop, but large enough to tell a coherent technical story. Typical starting points are a subdomain, a bounded context, or the area between two pivotal domain events from a big picture workshop.

The starting point is made visible on the board and is usually a central event or decision. It marks the boundary of the exploration: from here, we want to understand how we get from one business state to the next.

The goal of this first phase is clarity, not completeness.

6.4.3 Chaotic exploration of rules

Once the scope has been set, it's time to focus on the functional specifics. Which rules determine how the system behaves? These rules are at the heart of the workshop. They are collected on large yellow sticky notes, usually freely and without any particular order.

Each rule describes a condition, criterion, or decision that is essential to the functionality. It is formulated in everyday domain language, without resorting to technical terms. The rule here is: it is better to collect too many rules than to structure them too early.

The board should make thinking visible, not enforce order.

At this point, an initial connection to test-driven development can already be established. Each rule noted on a yellow sticky note can be considered a test hypothesis. In early implementation, this can result in a unit test that checks

whether this rule is fulfilled in the code. In this way, a workshop artifact becomes a building block of technical quality assurance.

6.4.4 Grouping the rules - making consistency boundaries visible

Once enough rules have been collected, the structuring phase begins. The rules are now grouped according to the criterion of which ones must always remain consistent with each other. These groups represent functional consistency boundaries. They are what is later referred to as aggregates in tactical DDD, but here they are more consciously described as consistent rule sets.

Grouping is not a mechanical step, but rather an in-depth discussion. Subject matter experts and developers compare their mental models: Which rules belong together from a subject matter perspective? Which ones are dependent on each other? Which ones can exist independently of each other?

At this point, the perspective shifts from loose rules to system logic. The resulting groups are semantic units. They form the core of what will later be implemented in the domain layer as aggregate roots, entities, and value objects. Their names are crucial, they become elements of the ubiquitous language and should appear unchanged in the code later on.

6.4.5 Adding Commands and Events

Once the grouped rules are in place, they are linked to interaction. The question is: How is this rule triggered, and what happens when it is executed?

The answers lead to commands and domain events. A command is an action, a call that activates a rule. An event is the observation of the result. Commands are written on blue sticky notes, events on orange sticky notes, and attached directly to the corresponding rule groups.

This creates the backbone of the model: a sequence of causes and effects, a story that shows the business logic in motion.

For test-driven development, this means that every combination of command and event becomes a test case. A test describes the expected effect of a business call: "When command X is executed, event Y occurs." The test cases document the behavior of the aggregates and thus the behavior of the system.

6.4.6 Policies and Read Models – wiring the system

The final step is to connect the system. Now it's all about the flow of reactions: how individual parts of the domain interact with each other. This is where policies and read models come into play.

Policies describe orchestrating rules that respond to events. They connect events with new commands in an if (event), then (commands) semantics. When a specific event occurs, a functional response is automatically triggered. This response can be synchronous or asynchronous, internal or cross-functional, but it is always an expression of a business rule. In later architecture, policies are often candidates for application or domain services.

Read models, on the other hand, are the perspective of evaluation. They are not used for control, but for observation. A read model projects domain events into a form that is understandable for users or external systems such as a map, a report, a display. They can also play an exciting role in CQRS-based architectures.

At the end of this phase, the board is a complete, functionally coherent representation of the context. All aggregates, commands, events, policies, and read models are interconnected. The system has a recognizable topology: a network of meaning, responsibility, and communication.

6.4.7 Result – A system that explains itself

A successful Design Level EventStorming does not end with a colorful board, but with a shared understanding. Participants have not only gathered rules, but also learned how these rules are translated into architecture, code, and language. The resulting systems are thus more understandable, testable, and expandable. They

are not based on assumptions, but on verified meaning. This combination of workshop, test, and architecture reveals the true strength of tactical Domain-Driven Design: it combines thinking, acting, and developing into a coherent practice.

6.5 Summary and outlook

Tactical Domain-Driven Design is the art of translating domain expertise into structure. It combines the building blocks of the model with architectural principles and practical methods to form a coherent whole. The patterns—entities, value objects, aggregates, domain events, services, repositories, and factories—are the basic vocabulary of this language. Architectures such as layered, hexagonal, CQRS, or event sourcing are its grammar, and deployment options are its forms of expression.

Those who practice DDD at the tactical level build software that speaks the language of the domain. The goal is not code that merely works, but a system that is understood, evolves with the domain, and accepts change as part of its nature. Tactical DDD is therefore not the end of architecture, but its living core: the constant translation of knowledge into design, design into code, and code back into shared understanding.

7 Conclusion: Do's and Don'ts in Domain-Driven Design

Domain-driven design is not a dogma, but rather an attitude that shapes thinking, collaboration, and architectural decisions. Nevertheless, there are typical pitfalls that almost every team experiences at some point and equally typical success factors that determine whether a project or product will be successful. This concluding chapter summarizes the most important dos and don'ts as a hands-on guidance for practitioners.

7.1 Business first, technology later

DDD always starts with the **problem space**. If you jump too quickly into frameworks, technologies, or architecture diagrams, it's easy to lose sight of the actual "why."

Do

- Start with the domain: customer needs, business processes, goals, value propositions.
- Utilize collaborative modeling workshops such as *EventStorming* or *Domain Storytelling* to gain a shared understanding of what is truly relevant.
- Use helpers such as the *Bounded Context Design Canvas* to consciously draw functional and technical boundaries.

Don't

- Don't get carried away by hype technologies or "best practices."
- Don't split up bounded contexts along technical layers (e.g., "front-end context," "back-end context").
- Don't confuse "code structure" with "domain structure."

7.2 Take the ubiquitous language seriously

Ubiquitous language is the central anchor between business and technology. It arises in conversations, not in code generators.

Do

- Consistently use the terminology of experts in code, tests, and documentation.
- Keep terms visible, for example, on the bounded context canvas or in glossaries.
- Question ambiguous or overloaded terms ("customer," "application," "case," etc.).

Don't

- Automatically translate domain terminology into technical synonyms ("CustomerDTO," "ClientEntity").
- Use multiple competing languages in a bounded context.
- Leave decisions about terms uncommented, they are deliberate decisions.

7.3 Design models in a collaborative way

Good models are created **through dialogue**, not by working alone.

Do

- Plan regular, short modeling sessions, not just big one-off workshops.
- Visualize: whiteboard, Miro, sticky notes.
- Keep models alive. They are tools for thinking, not artifacts for archiving.
- Combine *Big Picture EventStorming* for overview with *Design Level EventStorming* for detailed work.

Don't

- Hand models over as "finished" from one stakeholder to the next.
- Let meetings degenerate into lectures. Everyone should be allowed to stick, write, and move things around.
- Hide models in highly specialized tools.

7.4 Draw small, coherent boundaries

A bounded context is as large as what can be described in a consistent language.

Do

- Split contexts based on functional cohesion, not based on existing teams, technologies, or databases.
- Keep **cohesion high** and **coupling low**.
- If there are too many rules or terms, check whether the context has become too large.

Don't

- Create "mega contexts" that contain everything and generalize a lot.
- Centralize data models ("business partners," "master data") without a clear functional purpose.
- Couple contexts via shared databases, use *Published Language* or *Events* instead.

7.5 Think and learn iteratively

DDD is a learning process. Models mature, boundaries shift, language changes.

Do

- Work in small evolutions instead of big bang refactorings.
- Use TDD to gradually translate rules into code.
- Document open questions and assumptions ("Assumptions," "Open Questions") visibly on the bounded context design canvas.
- Reflect regularly: Do our models still correspond to reality?

Don't

- Search for "the perfect model right away" – the first iteration can only ever have the quality rating "they always tried very hard."
- Wait with implementation until everything seems clear, learning happens through doing.
- See refactoring as error correction, it is a central part of learning.

7.6 Designing technical resilience

Software is never error-free but it can be **resilient** if business logic is deliberately decoupled.

Do

- Implement "graceful degradation": systems should continue to provide valuable responses even in the event of partial failures.
- Decouple subdomains using domain events and policies instead of synchronous call chains.
- Practice how failures or delayed events affect business processes.

Don't

- Build rigid dependencies between contexts ("This one calls that one, which calls the other one").
- Hide errors behind technical retries, understand their functional significance.

7.7 Adapt architectures to the problem

Patterns are utilities, not laws.

Do

- Select architectural styles (monolith, hexagon, microservice, event-driven) based on context.
- In core domains: high quality, in-house development, test-driven models.
- In generic domains: custom of the shelf software, SaaS, or low-code is often sufficient.

Don't

- Use hexagonal architecture because it sounds fashionable.
- Pursue microservices when team size, quality requirements or business logic do not justify them.
- Abuse patterns as an end in themselves.

7.8 Make responsibility visible

DDD is socio-technical: architecture and organization must fit together.

Do

- Align teams along bounded contexts (Team Topologies: Stream-aligned Teams).
- Make upstream/downstream relationships explicit (Context Map).
- Clarify ownership before defining interfaces.

Don't

- Hide team dependencies behind technical APIs.
- Expect architecture alone to solve organizational problems.
- Ignore communication channels: they shape software just as much as code (Conway's Law).

7.9 Maintain pragmatism

DDD requires discipline, but also a healthy dose of common sense.

Do

- Use DDD where complexity and added value are high (core domain).
- Keep things lightweight in supporting and generic areas.
- Use DDD vocabulary as a thinking tool, not as a label.

Don't

- Use "full DDD" everywhere.
- Discuss pattern orthodoxy instead of business value.
- Confuse *creating complexity* with *mastering complexity*.

7.10 Summary

Domain-Driven Design is not a recipe book, but rather a continuous dialogue between domain expertise and engineering. Those who take the dos and don'ts described here to heart lay the groundwork for this: a language that connects, models that make sense, and architectures that allow for change.

Ultimately, **DDD is not a goal, but a daily practice.** Those who live it learn how models create value: not through dogma, but through understanding, collaboration, and continuous improvement.

8 Sources and references

DDD Crew on GitHub: https://www.github.com/ddd-crew

Brandolini, Alberto (2015): Introducing Event Storming, Leanpub https://leanpub.com/introducing_eventstorming

Dahan, Udi: Clarified CQRS http://udidahan.com/2009/12/09/clarified-cqrs/

Domain Storytelling Homepage: http://www.domainstorytelling.org/

Evans, Eric (2003): Domain-driven Design, Addison Wesley

Evans, Eric (2016): Model Exploration Whirlpool https://domainlanguage.com/ddd/whirlpool/

Khononov, Vlad (2021): Learning Domain-Driven Design, O'Reilly

Manifesto for Agile Software Development https://agilemanifesto.org/

Patton, Jeff (2014): User Story Mapping: Discover the Whole Story, Build the Right Product, O'Reilly and Associates

Principles behind the Agile Manifesto https://agilemanifesto.org/principles.html

Stevens, Meyers, Constantine (1974): Structured Design https://dl.acm.org/doi/10.5555/1241515.1241533

Vaughn, Vernon (2013): Implementing Domain-driven Design, Addison Wesley

9 About us

INNOQ is a technology consulting company. Honest consulting, innovative thinking, and a passion for software development means: We deliver successful software solutions, infrastructure and products.

We specialize in the following areas:

- Architecture Strategy
- Software Architecture and Development
- Data & AI
- IT Security
- Development of Digital Products
- Digital Platforms and Infrastructures
- Knowledge Transfer, Coaching and Trainings

With around 150 employees across offices in Germany and Switzerland, we support companies and organizations in designing and implementing complex initiatives and in improving existing software systems.

We are actively involved in open-source projects and the iSAQB® e.V., and we share our knowledge and experience at conferences and meetups, as well as through numerous books and professional publications.

Visit us at **www.innoq.com**

About the author

Michael Plöd

Michael is a Fellow at INNOQ. His current areas of focus are Domain-Driven Design, Team Topologies, socio-technical architectures, and the transformation of IT organisations towards collaboration and loosely coupled teams. Michael is the author of the book “Hands-on Domain-driven Design - by example” on Leanpub, translator of the book Team Topologies into German for O’Reilly, contributor/committer for DDD-Crew on GitHub, and a regular speaker at national and international conferences.

This primer provides a systematic overview of modern DDD. From collaborative modeling techniques such as Event Storming to strategic patterns such as Bounded Contexts to tactical implementation patterns. We show how DDD acts as a bridge between business and technology, what pitfalls lurk in practice, and how the approach connects with current trends such as data mesh or AI-driven domains.

For software architects, developers, product owners, and anyone who wants to understand and design complex software systems. Compact, practical, and up to date with the latest developments in the DDD community.

innoq.com