CHAPTER 1

Introducing Domain-Driven Design

As a developer, you may think that your job is to write code.

I disagree. A developer's job is to solve a problem through software, and coding is just one aspect of software development. Good design and communication are just as important, if not more so.

If you think of software development as a pipeline with an input (requirements) and an output (the final deliverable), then the "garbage in, garbage out" rule applies. If the input is bad (unclear requirements or a bad design), then no amount of coding can create a good output.

In the first part of this book we'll look at how to minimize the "garbage in" part by using a design approach focused on clear communication and shared domain knowledge: *domain-driven design*, or *DDD*.

In this chapter, we'll start by discussing the principles of DDD and by showing how they can be applied to a particular domain. DDD is a large topic, so we won't be exploring it in detail (for more detailed information on DDD, visit dddcommunity.org[1]). However, by the end of this chapter you should at least have a good idea of how domain-driven design works and how it is different from database-driven design and object-oriented design.

Domain-driven design is not appropriate for all software development, of course. There are many types of software (systems software, games, and so on) that can be built using other approaches. However, it is particularly useful for business and enterprise software, where developers have to collaborate with other nontechnical teams, and that kind of software will be the focus of this book.

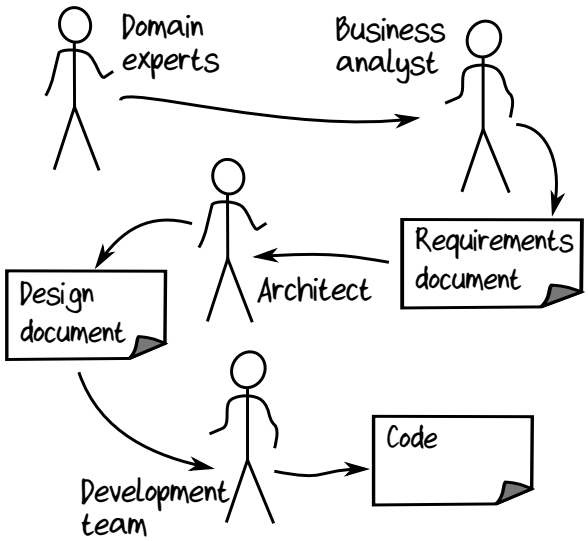
1. http://dddcommunity.org

The Importance of a Shared Model

Before attempting to solve a problem it's important that we understand the problem correctly. Obviously, if our understanding of the problem is incomplete or distorted, then we won't to be able to provide a useful solution. And sadly, of course, it's the developers' understanding, not the domain experts' understanding, that gets released to production!

So how can we ensure that we, as developers, *do* understand the problem?

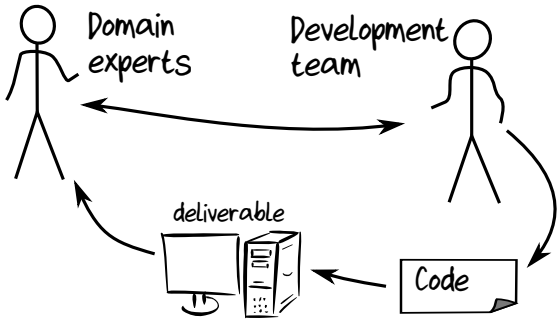
Some software development processes address this by using written specifications or requirements documents to try to capture all the details of a problem. Unfortunately, this approach often creates distance between the people who understand the problem best and the people who will implement the solution. We'll call the latter the "development team," by which we mean not just developers but also UX and UI designers, testers, and so on. And we'll call the former "domain experts." I won't attempt to define "domain expert" here—I think you'll know one when you see one!



In a children's game called "Telephone," a message is whispered from person to person along a chain of people. With each retelling the message gets more and more distorted, with comic results.

It's not so funny in a real-world development project. A mismatch between the developer's understanding of the problem and the domain expert's understanding of the problem can be fatal to the success of the project.

A much better solution is to eliminate the intermediaries and encourage the domain experts to be intimately involved with the development process, introducing a feedback loop between the development team and the domain expert. The development team regularly delivers something to the domain expert, who can quickly correct any misunderstandings for the next iteration.

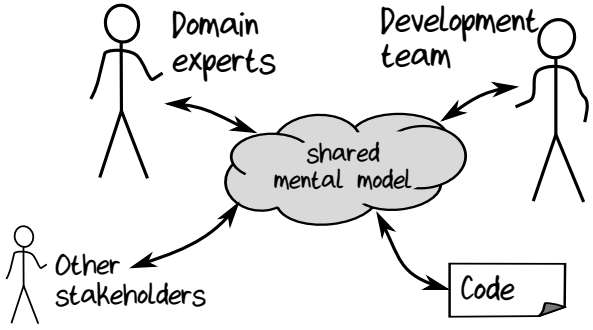


This kind of iterative process is at the core of "agile" development processes.

However, even this approach has its problems. The developer acts as a translator, translating the domain expert's mental model into code. But as in any translation, this process can result in distortion and loss of important subtleties. If the code doesn't quite correspond to the concepts in the domain, then future developers working on the codebase without input from a domain expert can easily misunderstand what's needed and introduce errors.

But there is a third approach. What if the domain experts, the development team, other stakeholders, and (most importantly) the source code itself all share the *same* model? In this case, there is no translation from the domain expert's requirements to the code. Rather, the code is designed to reflect the shared mental model directly.

And that is the goal of domain-driven design.



Aligning the software model with the business domain has a number of benefits:

- *Faster time to market.* When the developer and the codebase share the same model as the person who has the problem, the team is more likely to develop an appropriate solution quickly.
- *More business value.* A solution that is accurately aligned with the problem means happier customers and less chance of going offtrack.
- *Less waste.* Clearer requirements means less time wasted in misunderstanding and rework. Furthermore, this clarity often reveals which components are high value so that more development effort can be focused on them and less on the low-value components.
- *Easier maintenance and evolution.* When the model expressed by the code closely matches the domain expert's own model, making changes to the code is easier and less error-prone. Furthermore, new team members are able to come up to speed faster.

The Insanely Effective Delivery Machine

Dan North, the well-known developer and promoter of Behavior-Driven Development, described his experience with a shared mental model in his talk "Accelerating Agile." He joined a small team at a trading firm, which he described as the most insanely effective delivery machine he'd ever been a part of. In that firm, a handful of programmers produced state-of-the-art trading systems in weeks rather than months or years.

One of the reasons for the success of this team was that the developers were trained to be traders alongside the real traders. That is, they became domain experts themselves. This in turn meant that they could communicate very effectively with the traders, due to the shared mental model, and build exactly what their domain experts (the traders) wanted.

So we need to create a shared model. How can we do this? The domain-driven design community has developed some guidelines to help us here. They are as follows:

- Focus on business events and workflows rather than data structures.
- Partition the problem domain into smaller subdomains.
- Create a model of each subdomain in the solution.
- Develop a common language (known as the "Ubiquitous Language") that is shared between everyone involved in the project and is used everywhere in the code.

Let's look at these in turn.

Understanding the Domain Through Business Events

A DDD approach to gathering requirements will emphasize building a shared understanding between developers and domain experts. But where should we start in order to develop this understanding?

Our first guideline says to focus on business events rather than data structures. Why is that?

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

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

Whatever it is, it's important to capture it as part of the design. We call these things *Domain Events*.

Domain Events are the starting point for almost all of the business processes we want to model. For example, "new order form received" is a Domain Event that will kick off the order-taking process.

Domain Events are always written in the past tense—something happened—because it's a fact that can't be changed.

Using Event Storming to Discover the Domain

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

In Event Storming, you bring together a variety of people (who understand different parts of the domain) for a facilitated workshop. The attendees should include not just developers and domain experts but all the other stakeholders who have an interest in the success of the project: as event stormers like to say, "anyone who has questions and anyone who has answers." The workshop should be held in a room that has a lot of wall space, and the walls should be covered with paper or whiteboard material so that the participants can

post sticky notes or draw on them. At the end of a successful session, the walls will be covered with hundreds of these notes.

During the workshop, people write down business events on the sticky notes and post them on the wall. Other people may respond by posting notes summarizing the business workflows that are triggered by these events. These workflows, in turn, often lead to other business events being created. In addition, the notes can often be organized into a timeline, which may well trigger further discussion in the group. The idea is to get all the attendees to participate in posting what they know and asking questions about what they don't know. It's a highly interactive process that encourages everyone to be involved. For more detail on Event Storming in practice, see the *EventStorming* book by Alberto Brandolini,[2] the creator of this technique.

Discovering the Domain: An Order-Taking System

In this book, we'll take a realistic business problem—an order-taking system—and use it to explore design, domain modeling, and implementation.

Say that we are called in to help a small manufacturing company, Widgets Inc, to automate its order-taking workflow. Max, the manager at Widgets, explains:

> "We're a tiny company that manufactures parts for other companies: widgets, gizmos, and the like. We've been growing quite fast, and our current processes are not able to keep up. Right now, everything we do is paper-based, and we'd like to computerize all that so that our staff can handle larger volumes of orders. In particular, we'd like to have a self-service website so that customers can do some tasks themselves. Things like placing an order, checking order status, and so on."

Sounds good. So now what do we do? Where should we start?

The first guideline says "focus on business events," so let's use an event-storming session for that. Here's how one might start out at Widgets.

> You: "Someone start by posting a business event!"
>
> Ollie: "I'm Ollie from the order-taking department. Mostly we deal with orders and quotes coming in."
>
> You: "What triggers this kind of work?"
>
> Ollie: "When we get forms sent to us by the customer in the mail."
>
> You: "So the events would be something like 'Order form received' and 'Quote form received'?"

2. http://eventstorming.com

Ollie: "Yes. Let me put those up on the wall then."

Sam: "I'm Sam from the shipping department. We fulfill those orders when they're signed off."

You: "And how do you know when to do that?"

Sam: "When we get an order from the order-taking department."

You: "What would you call that as an event?"

Sam: "How about 'Order available'?"

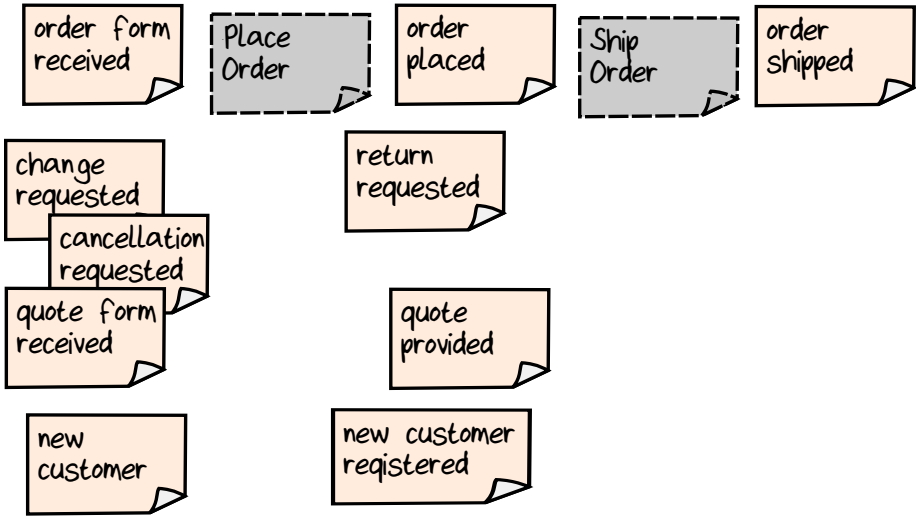
Ollie: "We call an order that's completed and ready to ship a 'Placed order.' Can we agree on using that term everywhere?"

Sam: "So 'Order placed' would be the event we care about, yes?"

You get the idea. After a while, we might have list of posted events like this:

- Order form received
- Order placed
- Order shipped
- Order change requested
- Order cancellation requested
- Return requested
- Quote form received
- Quote provided
- New customer request received
- New customer registered

Here's what the wall might look like at this point:



Some of the events have business workflows posted next to them, such as "Place order" and "Ship order," and we're beginning to see how the events connect up into larger workflows.

We can't cover a full event-storming session in detail, but let's look at some of the aspects of requirements gathering that Event Storming facilitates:

- *A shared model of the business*

As well as revealing the events, a key benefit of Event Storming is that the participants develop a shared understanding of the business, because everyone is seeing the same thing on the big wall. Just like DDD, Event Storming has an emphasis on communication and shared models and avoiding "us" vs. "them" thinking. Not only will attendees learn about unfamiliar aspects of the domain, but they might realize that their assumptions about other teams are wrong or perhaps even develop insights that can help the business improve.

- *Awareness of all the teams*

Sometimes it's easy to focus on just one aspect of the business—the one that you are involved in—and forget that other teams are involved and may need to consume data that you produce. If all the stakeholders are in the room, anyone who is being overlooked can speak out.

> "I'm Blake from the billing department. Don't forget about us. We need to know about completed orders too, so we can bill people and make money for the company! So we need to get an 'order placed' event as well."

- *Finding gaps in the requirements*

When the events are displayed on a wall in a timeline, missing requirements often become very clear:

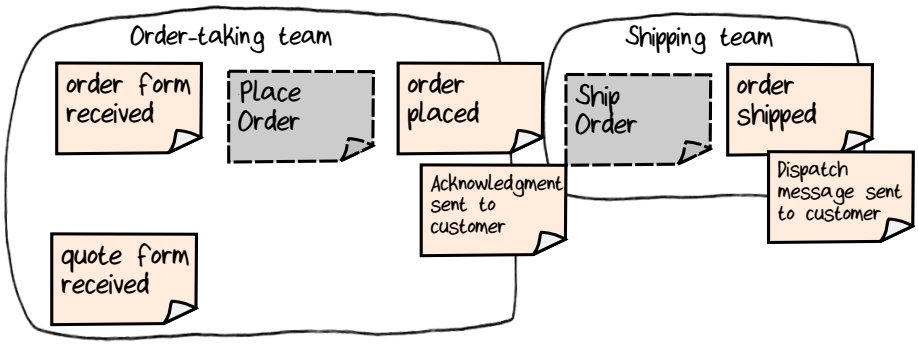
> Max: "Ollie, when you've finished preparing an order, do you tell the customer? I don't see that on the wall."
>
> Ollie: "Oh, yes. I forgot. When the order has been placed successfully, we send an email to the customer saying that we got it and are about to ship it. That's another event, I suppose: 'Order acknowledgment sent to customer'."

If the question doesn't have a clear answer, then the question itself should be posted on the wall as a trigger for further discussion. And if a particular part of the process creates debate or disagreement, don't treat it as a problem, treat it as an opportunity! You'll learn a lot by drilling into these areas. It's common for the requirements to be fuzzy at the beginning of a project, so documenting the questions and debate in this visible way makes it clear more work needs to be done, and it discourages starting the development process prematurely.

- *Connections between teams*

The events can be grouped in a timeline, which often makes it clear that one team's output is another team's input.

For example, when the order-taking team has finished processing an order, they need to signal that a new order has been placed. This "Order placed" event becomes the input for the shipping and billing teams:



The technical details of *how* the teams are connected is not relevant at this stage. We want to focus on the domain, not the pros and cons of message queues vs. databases.

- *Awareness of reporting requirements*

It's easy to focus only on processes and transactions when trying to understand the domain. But any business needs to understand what happened in the past—reporting is always part of the domain! Make sure that reporting and other read-only models (such as view models for the UI) are included in the event-storming session.

Expanding the Events to the Edges

It is often useful to follow the chain of events out as far as you can, to the boundaries of the system. To start, you might ask if any events occur before the leftmost event.

> You: "Ollie, what triggers the 'Order form received' event? Where does that come from?"
>
> Ollie: "We open the mail every morning, and the customers send in order forms on paper, which we open up and classify as orders or quotes."
>
> You: "So it looks like we need a 'Mail received' event as well?"

Workflows, Scenarios, and Use Cases

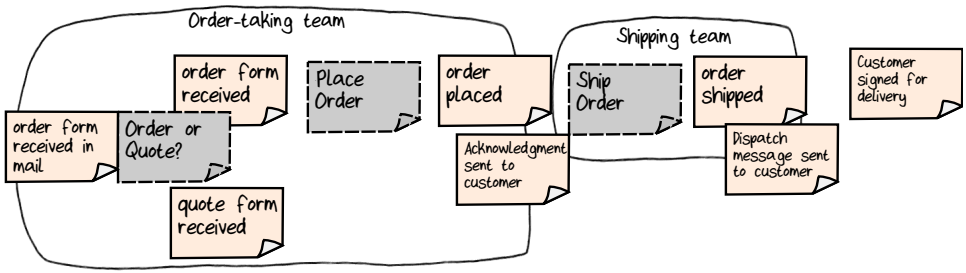
We have many different words to describe business activities: "workflows," "scenarios," "use cases," "processes," and so on. They're often used interchangeably; but in this book, we'll try to be a bit more precise.

- A *scenario* describes a goal that a customer (or other user) wants to achieve, such as placing an order. It is similar to a "story" in agile development. A *use case* is a more detailed version of a scenario, which describes in general terms the user interactions and other steps that the user needs to take to accomplish a goal. Both *scenario* and *use case* are user-centric concepts, focused on how interactions appear from the user's point of view.
- A *business process* describes a goal that the business (rather than an individual user) wants to achieve. It's similar to a scenario but has a business-centric focus rather than a user-centric focus.
- A *workflow* is a detailed description of part of a business process. That is, it lists the exact steps that an employee (or software component) needs to do to accomplish a business goal or subgoal. We'll limit a workflow to what a single person or team can do, so that when a business process is spread over multiple teams (as the ordering process is), we can divide the overall business process into a series of smaller workflows, which are then coordinated in some way.

In the same way, we might extend the events on the shipping side of the business.

> You: "Sam, are there any possible events *after* you ship the order to the customer?"
>
> Sam: "Well, if the order is "Signed for delivery," we'll get a notification from the courier service. So let me add a 'Shipment received by customer' event."

Extending the events out as far as you can in either direction is another great way of catching missing requirements. You might find that the chain of events ends up being longer than you expect.



Notice that the domain expert is talking about paper forms and printed mail. The system that we want to replace this with will be computerized, but we can learn a lot by thinking about paper-based systems in terms of workflow, prioritization, edge cases, and so on. Let's focus on understanding the domain for now; only when we understand it thoroughly should we think about how to implement a digital equivalent.

Indeed, in many business processes the whole paper vs. digital distinction is irrelevant—understanding the high-level concepts of the domain does not depend on any particular implementation at all. The domain of accounting is a good example; the concepts and terminology have not changed for hundreds of years.

Also, when converting a paper-based system to a computerized system, there's often no need to convert all of it at once. We should look at the system as a whole and start by converting only the parts that would benefit most.

Documenting Commands

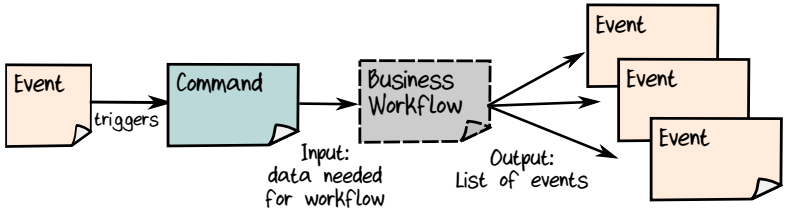
Once we have a number of these events on the wall, we might ask, "What made these Domain Events happen?" Somebody or something wanted an activity to happen. For example, the customer wanted us to receive an order form, or your boss asked you to do something.

We call these requests *commands* in DDD terminology (not be confused with the Command pattern used in OO programming). Commands are always written in the imperative: "Do this for me."

Of course, not all commands actually succeed—the order form might have gotten lost in the mail, or you're too busy with something more important to help your boss. But if the command does succeed, it will initiate a workflow that in turn will create corresponding Domain Events. Here are some examples:

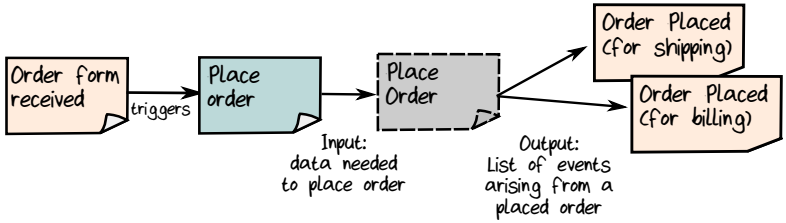
- If the command was "Make *X* happen," then, if the workflow made *X* happen, the corresponding Domain Event would be "*X* happened."
- If the command was "Send an order form to Widgets Inc," then, if the workflow sent the order, the corresponding Domain Event would be "Order form sent."
- Command: "Place an order"; Domain Event: "Order placed."
- Command: "Send a shipment to customer ABC"; Domain Event: "Shipment sent."

In fact, we will try to model most business processes in this way. An event triggers a command, which initiates some business workflow. The output of the workflow is some more events. And then, of course, those events can trigger further commands.



This way of thinking about business processes—a pipeline with an input and some outputs—is an excellent fit with the way that functional programming works, as we will see later.

Using this approach, then, the order-taking process looks like this:



For now, we'll assume that every command succeeds and the corresponding event happens. Later on, in Chapter 10, *Implementation: Working with Errors*, on page 191, we'll see how to model failure—how to handle the cases when things go wrong and commands do not succeed.

By the way, not *all* events need be associated with a command. Some events might be triggered by a scheduler or monitoring system, such as `MonthEndClose` for an accounting system or `OutOfStock` for a warehouse system.

Partitioning the Domain into Subdomains

We now have a list of events and commands, and we have a good understanding of what the various business processes are. But the big picture is still quite chaotic. We'll have to tame it before we start writing any code.

This brings us to our second guideline: "Partition the problem domain into smaller subdomains." When faced with a large problem, it's natural to break it into smaller components that can be addressed separately. And so it is here. We have a large problem: organizing the events around order taking. Can we break it into smaller pieces?

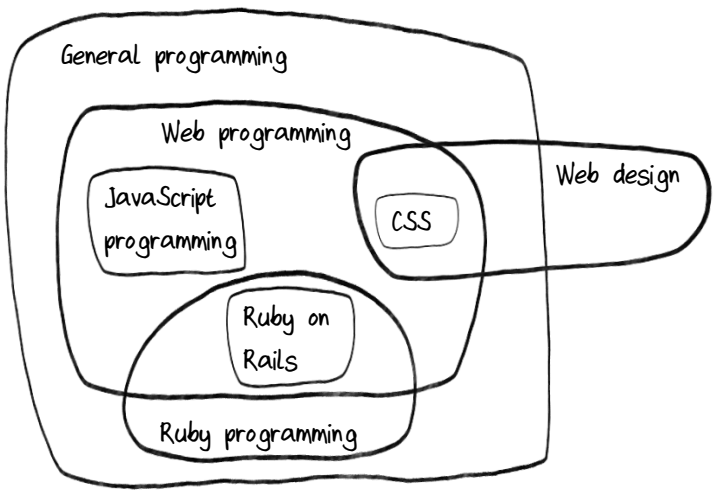
Yes, we can. It's clear that various aspects of the "order-taking process" can be separated: the order taking, the shipping, the billing, and so on. As we know, the business already has separate departments for these areas, and that's a pretty strong hint that we can follow that same separation in our design. We will call each of these areas a *domain*.

Now *domain* is a word with many meanings, but in the world of domain-driven design, we can define a "domain" as "an area of coherent knowledge." Unfortunately that definition is too vague to be useful, so here's an alternative definition of a domain: a "domain" is just that which a "domain expert" is expert in! This is much more convenient in practice: rather than struggling to provide a dictionary definition of what "billing" means, we can just say that "billing" is what people in the billing department—the domain experts—do.

We all know what a "domain expert" is; as programmers we ourselves are often experts in a number of domains. For example, you could be an expert in the use of a particular programming language or in a particular area of programming, such as games or scientific programming. And you might have knowledge of areas such as security or networking or low-level optimizations. All these things are "domains."

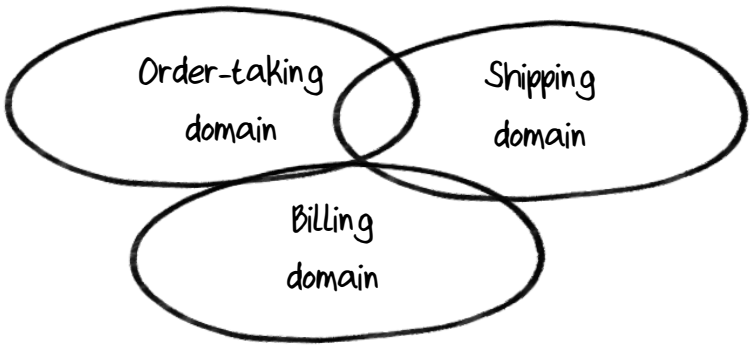
Within a domain might be areas that are distinctive as well. We call these *subdomains*—a smaller part of a larger domain that has its own specialized knowledge. For example, "web programming" is a subdomain of "general programming." And "JavaScript programming" is a subdomain of web programming (at least, it used to be).

Here's a diagram showing some programming-related domains:



You can see that domains can overlap. For example, the "CSS" subdomain could be considered part of the "web programming" domain but also part of the "web design" domain. So we must be careful when partitioning a domain into smaller parts: it's tempting to want clear, crisp boundaries, but the real world is fuzzier than that.

If we apply this domain-partitioning approach to our order-taking system, we have something like this:



The domains overlap a little bit. An order-taker must know a little bit about how the billing and shipping departments work, a shipper must know a little bit about how the order-taking and billing departments work, and so on.

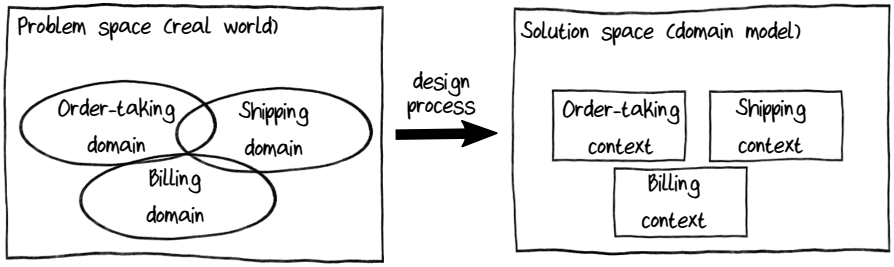
As we have stressed before, if you want be effective when developing a solution, you need become a bit of a domain expert yourself. That means that, as developers, we'll need to make an effort to understand the domains above more deeply than we have done so far.

But let's hold off on that for now and move on to the guidelines for creating a solution.

Creating a Solution Using Bounded Contexts

Understanding the problem doesn't mean that building a solution is easy. The solution can't possibly represent *all* the information in the original domain, nor would we want it to. We should only capture the information that is relevant to solving a particular problem. Everything else is irrelevant.

We therefore need to create a distinction between a "problem space" and a "solution space," and they must be treated as two different things. To build the solution we will create a *model* of the problem domain, extracting only the aspects of the domain that are relevant and then re-creating them in our solution space as shown in the figure on page 17.



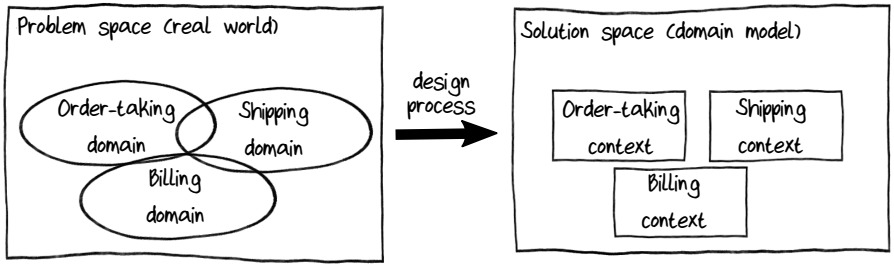
In the solution space, you can see that the domains and subdomains in the problem space are mapped to what DDD terminology calls *bounded contexts*—a kind of subsystem in our implementation. Each bounded context is a mini domain model in its own right. We use the phrase *bounded context* instead of something like *subsystem* because it helps us stay focused on what's important when we design a solution: being aware of the context and being aware of the boundaries.

Why *context*? Because each context represents some specialized knowledge in the solution. Within the context, we share a common language and the design is coherent and unified. But, just as in the real world, information taken out of context can be confusing or unusable.

Why *bounded*? In the real world, domains have fuzzy boundaries, but in the world of software we want to reduce coupling between separate subsystems so that they can evolve independently. We can do this using standard software practices, such as having explicit APIs between subsystems and avoiding dependencies such as shared code. This means, sadly, that our domain model will never be as rich as the real world, but we can tolerate this in exchange for less complexity and easier maintenance.

A domain in the problem space does not always have a one-to-one relationship to a context in the solution space. Sometimes, for various reasons, a single domain is broken into multiple bounded contexts—or more likely—multiple domains in the problem space are modeled by only one bounded context in the solution space. This is especially common when you need to integrate with a legacy software system.

For example, in an alternate world, Widgets Inc might already have installed a software package that did order taking and billing together in one system. If you needed to integrate with this legacy system, you would probably need to treat it as a single bounded context, even though it covers multiple domains as shown in the figure on page 18.



However you partition the domain, it's important that each bounded context have a clear responsibility, because when we come to implement the model, a bounded context will correspond exactly to some kind of software component. The component could be implemented as a separate DLL, or as a standalone service, or just as a simple namespace. The details don't matter right now, but getting the partitioning right is important.

Getting the Contexts Right

Defining these bounded contexts sounds straightforward, but it can be tricky in practice. Indeed, one of the most important challenges of a domain-driven design is to get these context boundaries right. This is an art, not a science, but here are some guidelines that can help:

- *Listen to the domain experts.* If they all share the same language and focus on the same issues, they are probably working in the same subdomain (which maps to a bounded context).
- *Pay attention to existing team and department boundaries.* These are strong clues to what the business considers to be domains and subdomains. Of course, this is not always true: sometimes people in the same department are working at odds with each other. Conversely, people in different departments may collaborate very closely, which in turn may mean they're working in the same domain.
- *Don't forget the "bounded" part of a bounded context.* Watch out for scope creep when setting boundaries. In a complex project with changing requirements, you need to be ruthless about preserving the "bounded" part of the bounded context. A boundary that is too big or too vague is no boundary at all. As the saying goes, "Good fences make good neighbors."
- *Design for autonomy.* If two groups contribute to the same bounded context, they might end up pulling the design in different directions as it evolves. Think of a three-legged race: two runners tied at the leg are much

slower than two runners free to run independently. And so it is with a domain model. It's always better to have separate and autonomous bounded contexts that can evolve independently than one mega-context that tries to make everyone happy.

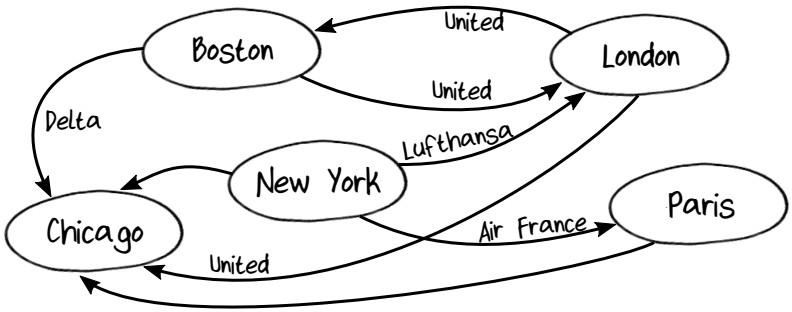
- *Design for friction-free business workflows.* If a workflow interacts with multiple bounded contexts and is often blocked or delayed by them, consider refactoring the contexts to make the workflow smoother, even if the design becomes "uglier." That is, always focus on business and customer value rather than any kind of "pure" design.

Finally, no design is static, and any model must need to evolve over time as the business requirements change. We will discuss this further in Chapter 13, *Evolving a Design and Keeping It Clean*, on page 265, where we will demonstrate various ways to adapt the order-taking domain to new demands.

Creating Context Maps

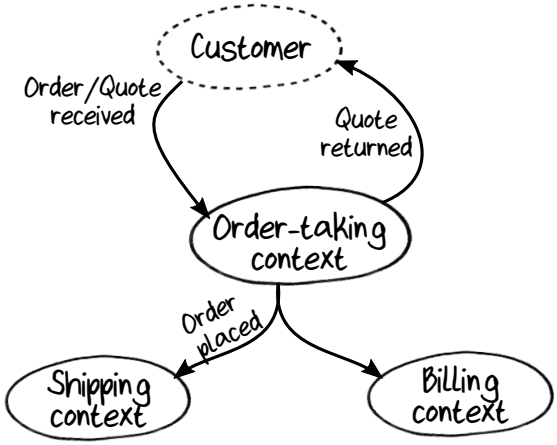
Once we have defined these contexts, we need a way to communicate the interactions between them—the big picture—without getting bogged down in the details of a design. In DDD terminology, these diagrams are called *Context Maps*.

Think of a route map used for traveling. A route map doesn't show you every detail: it focuses only on the main routes so that you can plan your journey. For example, here's a sketch of an airline route map:



This diagram doesn't show the details of each city, just the available routes between each city. The map's only purpose is to help you plan your flights. If you want to do something different, such as drive around New York, you're going to need a different map (and some blood pressure pills).

In the same way, a context map shows the various bounded contexts and their relationships at a high level. The goal is not to capture every detail but to provide a view of the system as a whole. For example, this is what we have so far for the order-taking system as shown in the figure on page 20.



In making this map, we are not concerned with the internal structure of the shipping context, only that it receives data from the order-taking context. We say informally that the shipping context is *downstream* and the order-taking context is *upstream*.

Obviously the two contexts will need to agree on a shared format for the messages that they exchange. In general, the upstream context has more influence over the format, but sometimes the downstream context is inflexible (such as working with a legacy system); and either the upstream context must adapt to that, or some sort of translator component will be needed as an intermediary. (This is discussed further in *Contracts Between Bounded Contexts*, on page 48.)

Finally, it's worth pointing out that in our design we can fit everything into one map (so far). In more complex designs, you'd naturally want to create a series of smaller maps, each focusing on specific subsystems.

Focusing on the Most Important Bounded Contexts

We have a few obvious bounded contexts at this point, and we may find that we discover more as we work with the domain. But are they all equally important? Which ones should we focus on when we start development?

Generally, some domains *are* more important than others. These are the core domains—the ones that provide a business advantage, the ones that bring in the money.

Other domains may be required but are not core. These are called *supportive* domains, and if they are not unique to the business they are called *generic* domains.

For example, for Widgets Inc, the order-taking and shipping domains might be core, because their business advantage is their excellent customer service. The billing domain would be considered as supportive, and delivery of the shipments could be considered generic, which means they can safely outsource it.

Of course, reality is never as simple. Sometimes the core domain is not what you might expect. An e-commerce business might find that having items in stock and ready to ship is critical to customer satisfaction, in which case inventory management might become a core domain, just as important to the success of the business as an easy-to-use website.

Sometimes there's no consensus about what is the most important domain; each department may think that its domain is the most important. And sometimes, the core domain is simply whatever your client wants you to work on.

In all cases though, it is important to prioritize and not to attempt to implement all the bounded contexts at the same time—that often leads to failure. Focus instead on those bounded contexts that add the most value, and then expand from there.

Creating a Ubiquitous Language

We said earlier the code and the domain expert must share the same model.

That means that things in our design must represent real things in the domain expert's mental model. That is, if the domain expert calls something an "order," then we should have something called an `Order` in the code that corresponds to it and that behaves the same way.

And conversely, we should *not* have things in our design that do not represent something in the domain expert's model. That means no terms like `OrderFactory`, `OrderManager`, `OrderHelper`, and so forth. A domain expert wouldn't know what you meant by these words. Of course, some technical terms will have to occur in the codebase, but you should avoid exposing them as part of the *design*.

The set of concepts and vocabulary that is shared between everyone on the team is called the *Ubiquitous Language*—the "everywhere language." This is the language that defines the shared mental model for the business domain. And, as its name implies, this language should used *everywhere* in the project, not just in the requirements but in the design and, most importantly, in the source code.

The construction of the ubiquitous language is not a one-way process dictated by the domain expert, it is a collaboration between everyone on the team. Nor should you expect the ubiquitous language to be static: it's always a work in

progress. As the design evolves, be prepared to discover new terms and new concepts, and let the ubiquitous language evolve correspondingly. We'll see this happen in the course of this book.

Finally, it's important to realize that you often cannot have a single Ubiquitous Language that covers *all* domains and contexts. Each context will have a "dialect" of the Ubiquitous Language, and the same word can mean different things in different dialects. For example, "class" means one thing in the object-oriented programming domain but a completely different thing in the CSS domain. Trying to make a word like "Customer" or "Product" mean the same in different contexts can lead to complex requirements at best, and serious design errors at worst.

Indeed, our event-storming session demonstrates this exact issue. All the attendees used the word "order" when describing events. But we might well find that the shipping department's definition of "order" is subtly different definition than the billing department's definition. The shipping department probably cares about inventory levels, the quantity of items, and so on, while the billing department probably cares more about prices and money. If we use the same word "order" everywhere *without* specifying the context for its use, we might well run into some painful misunderstandings.

Summarizing the Concepts of Domain-Driven Design

We've been introduced to a lot of new concepts and terminology, so let's quickly summarize them in one place before moving on.

- A *domain* is an area of knowledge associated with the problem we are trying to solve, or simply, that which a "domain expert" is expert in.
- A *Domain Model* is a set of simplifications that represent those aspects of a domain that are relevant to a particular problem. The domain model is part of the solution space, while the domain that it represents is part of the problem space.
- The *Ubiquitous Language* is a set of concepts and vocabulary that is associated with the domain and is shared by both the team members and the source code.
- A *bounded context* is a subsystem in the solution space with clear boundaries that distinguish it from other subsystems. A bounded context often corresponds to a subdomain in the problem space. A bounded context also has its own set of concepts and vocabulary, its own dialect of the Ubiquitous Language.

- A *Context Map* is a high-level diagram showing a collection of bounded contexts and the relationships between them.
- A *Domain Event* is a record of something that happened in the system. It's always described in the past tense. An event often triggers additional activity.
- A *Command* is a request for some process to happen and is triggered by a person or another event. If the process succeeds, the state of the system changes and one or more Domain Events are recorded.

Wrapping Up

At the beginning of the chapter, we emphasized the importance of creating a shared model of the domain and solution—a model that is the same for the development team and the domain experts.

We then discussed four guidelines to help us do that:

- Focus on events and processes rather than data.
- Partition the problem domain into smaller subdomains.
- Create a model of each subdomain in the solution.
- Develop an "everywhere language" that can be shared between everyone involved in the project.

Let's see how we applied them to the order-taking domain.

Events and Processes

The event-storming session quickly revealed all the major *Domain Events* in the domain. For example, we learned that the order-taking process is triggered by receiving an order form in the mail, and that there are workflows for processing a quote, for registering a new customer, and so on.

We also learned that when the order-taking team finished processing an order, that event triggered the shipping department to start the shipping process and the billing department to start the billing process.

Many more events and processes could be documented, but we'll focus primarily on this one workflow for the rest of this book.

Subdomains and Bounded Contexts

It appears that we have discovered three *subdomains* so far: "Order Taking," "Shipping," and "Billing." Let's check our sense of this by using our "a domain is what a domain expert is expert in" rule.

You: "Hey Ollie, do you know how the billing process works?"

Ollie: "A little bit, but you should really ask the billing team if you want the details."

Billing is a separate domain? Confirmed!

We then defined three *bounded contexts* to correspond with these subdomains and created a *context map* that shows how these three contexts interact.

Which one is the *core* domain that we should focus on? We should really consult with Max the manager to decide where automation can add the most value, but for now, let's assume that we will implement the order-taking domain first. If needed, the output of the domain can be converted to paper documents so that the other teams can continue with their existing processes without interruption.

The Ubiquitous Language

So far we have terms like "order form," "quote," and "order," and no doubt we will discover more as we drill into the design. To help maintain a shared understanding, it would be a good idea to create a living document or wiki page that lists these terms and their definitions. This will help keep everyone aligned and help new team members get up to speed quickly.

What's Next?

We now have an overview of the problem and an outline of a solution, but we still have many questions that need answering before we can create a low-level design or start coding.

What happens, exactly, in the order-processing workflow? What are the inputs and outputs? Are there any other contexts that this workflow interacts with? How does the shipping team's concept of an "order" differ from the billing team's? And so on.

In the next chapter, we'll dive deeply into the order-placing workflow and attempt to answer these questions.