CHAPTER 2

# Understanding the Domain

In the previous chapter, we looked at the big picture—an overview of the domain and the key business events—and we divided the solution space into a number of bounded contexts. Along the way, we learned about domain-driven design and the importance of a shared model.

In this chapter, we're going to take one particular workflow and try to understand it deeply. What exactly triggers it? What data is needed? What other bounded contexts does it need to collaborate with?

We'll see that careful listening is a key skill in this process. We want to avoid imposing our own mental model on the domain.

## Interview with a Domain Expert

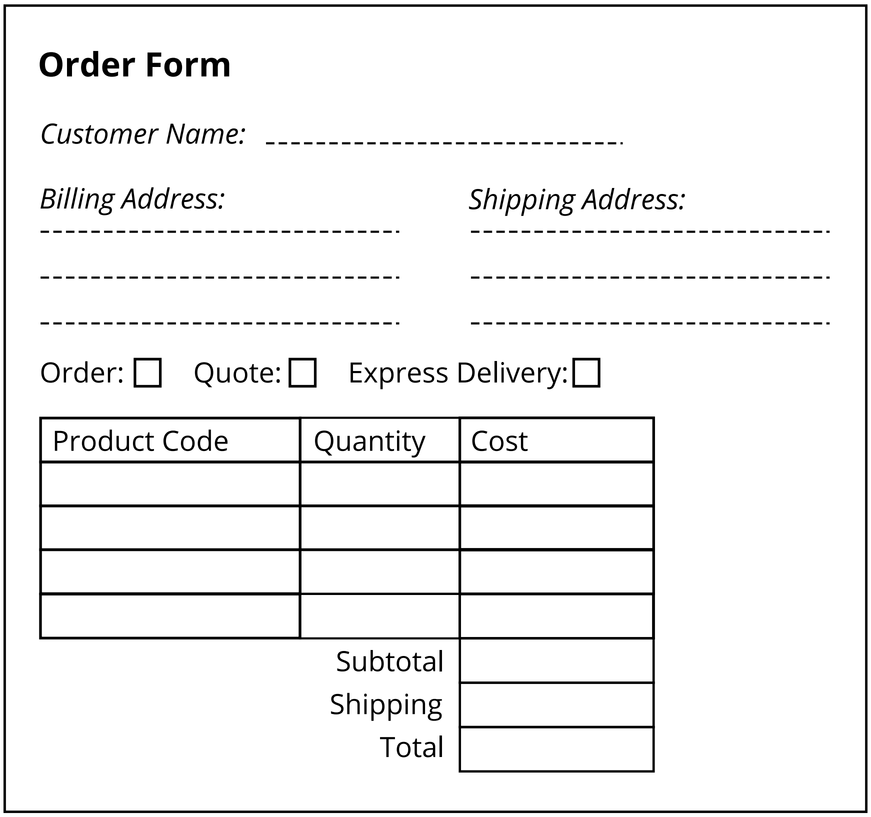
In order to get the understanding we want, let's do an in-depth interview with a domain expert: Ollie from the order-taking department.

Now, domain experts tend to be busy and generally can't spend too much time with developers. But one nice thing about the commands/events approach is that rather than needing all-day meetings, we can have a series of short interviews, each focusing on only one workflow, so a domain expert is more likely to be able to make time for this.

In the first part of the interview, we want to stay at a high level and focus only on the inputs and outputs of the workflow. This will help us avoid getting swamped with details that are not (yet) relevant to the design.

> You: "Ollie, let's talk about just one workflow, the order-placing process. What information do you need to start this process?"
>
> Ollie: "Well, it all starts with this piece of paper: the order form that customers fill out and send us in the mail. In the computerized version, we want the customer to fill out this form online."

Ollie shows you something that looks like this:



**Order Form**

*Customer Name:* ------------------------------

*Billing Address:*
------------------------------
------------------------------
------------------------------

*Shipping Address:*
------------------------------
------------------------------
------------------------------

Order: ☐ Quote: ☐ Express Delivery: ☐

| Product Code | Quantity | Cost |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | Subtotal | |
| | Shipping | |
| | Total | |

At this point you might think that this is a typical e-commerce model.

> You: "I see. So the customers will browse the product pages on the website, then click to add items to the shopping cart, and then check out?"
>
> Ollie: "No, of course not. Our customers already know exactly what they want to order. We just want a simple form where they can type in the product codes and quantities. They might order two or three hundred items at once, so clicking around in product pages to find each item first would be terribly slow."

This is an important lesson. You're supposed to be learning about the domain, so resist the urge to jump to conclusions about anything, such as (in this case) how the customers will use the system. Good interviewing means doing lots of listening! The best way to learn about a domain is to pretend you're an anthropologist and avoid having any preconceived notions. Ideally, we would do in-depth research (such as observing people at work, usability testing, and so on) before we commit to a design. In this case, though, we'll take the risk and skip these steps, trusting that Ollie understands the customer's needs well enough to represent them to us.

## Understanding the Non-functional Requirements

This would be a good time to take a step back and discuss the context and scale of the workflow.

> You: “Sorry, I misunderstood who the customer was. Let me get some more background information. For example, who uses this process and how often?”
>
> Ollie: “We’re a B2B company,[1] so our customers are other businesses. We have about 1000 customers, and they typically place an order every week.”
>
> You: “So about two hundred orders per business day. Does it ever get much busier than that, say in the holiday season?”
>
> Ollie: “No. It’s pretty consistent all year.”

This is good—we know that we don’t need to design for massive scale, nor do we have to design for spiky traffic. Now, what about customer expectations of the system?

> You: “And you say that the customers are experts?”
>
> Ollie: “They spend all day purchasing things, so yes, they are experts in that domain. They know what they want; they just need an efficient way to get it.”

This information affects how we think about the design. A system designed for beginners will often be quite different from a system designed for experts. If the customers are experts, then we don’t want to put barriers in their way or anything else that will slow them down.

> You: “What about latency? How quickly do they need a response?”
>
> Ollie: “They need an acknowledgment by the end of the business day. For our business, speed is less important than consistency. Our customers want to know that we will respond and deliver in a predictable way.”

These are typical requirements for a B2B application: needs like predictability, robust data handling, and an audit trail of everything that happens in case there are any questions or disputes.

## Understanding the Rest of the Workflow

Let’s keep going with the interview.

> You: “OK, what do you do with each form?”
>
> Ollie: “First we check that the product codes are correct. Sometimes there are typos or the product doesn’t exist.”
>
> You: “How you know if a product doesn’t exist?”

1. https://en.wikipedia.org/wiki/Business-to-business

> Ollie: "I look it up in the product catalog. It's a leaflet listing all the products and their prices. A new one is published every month. Look, here's the latest one sitting on my desk."

The product catalog sounds like another bounded context. We'll make a note to revisit it in detail later. For now, we'll skip it and just keep track of what this workflow needs from that context: the list of products and their prices.

> You: "And then?"
>
> Ollie: "Then we add up the cost of the items, write that into the Total field at the bottom, and then make two copies: one for the shipping department and one for the billing department. We keep the original in our files."
>
> You: "And then?"
>
> Ollie: "Then we scan the order and email it to the customer so that they can see the prices and the amount due. We call this an 'order acknowledgment.'"

OK, that makes sense so far. At some point you will want to go deeper into understanding how the validation is done and how the orders are transmitted to the other departments. One more question, though.

> You: "What are those boxes marked 'Quote' and 'Order' for?"
>
> Ollie: "If the 'Order' box is checked, then it's an order and if the 'Quote' box is checked, then it's a quote. Obviously."
>
> You: "So what's the difference between a quote and an order?"
>
> Ollie: "A quote is when the customer just wants us to calculate the prices but not actually dispatch the items. With a quote, we just add prices to the form and send it back to the customer. We don't send copies to the shipping and billing departments because there's nothing for them to do."
>
> You: "I see. Orders and quotes are similar enough that you use the same order form for both, but they have different workflows associated with them."

## Thinking About Inputs and Outputs

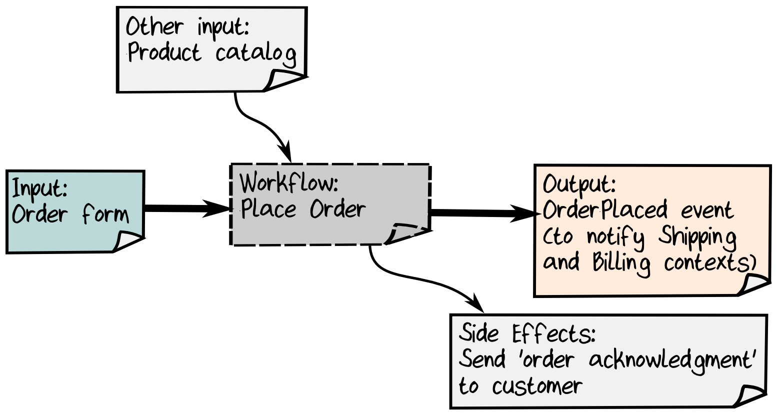
Let's pause to document what we've learned about the inputs and outputs of the workflow so far.

First, the input is clearly an order form (the exact definition of which we need to flesh out soon).

But what's the output? We've seen the concept of a "completed order" (based on the input but validated and with prices calculated). But that can't be the output, because we don't do anything with it directly. What about the "order acknowledgment" then? Could that be the output? Probably not. Sending the order acknowledgment is a side effect of the order-placing workflow, not an output.

The output of a workflow should always be the events that it generates, the things that trigger actions in other bounded contexts. In our case, the output of the workflow would be something like an "OrderPlaced" event, which is then sent to the shipping and billing contexts. (How the event actually gets to those departments is a discussion for later; it's not relevant to the design right now.)

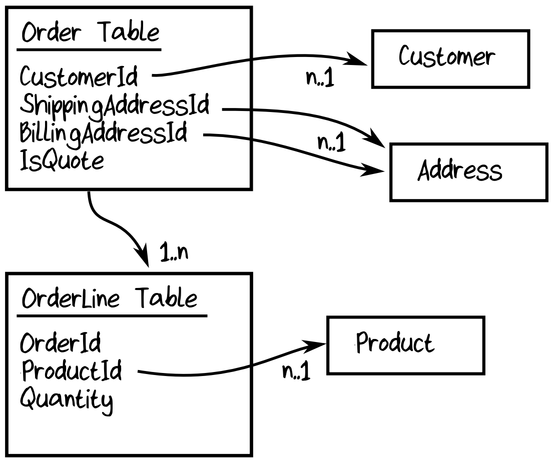
Let's diagram the "Place Order" workflow with its inputs and outputs:



## Fighting the Impulse to Do Database-Driven Design

At this point, if you are like most developers, you can't help but start sketching out a low-level design and implementation immediately.

For example, you might look at that order form and see that it consists of customer information, some addresses, a list of order lines, and so on.



If you have a lot of database experience, your first instinct might be to think about tables and the relationships between them. You might envision an Order table, an OrderLine table, and Customer, Address, and Product tables. And then you'll probably want to describe the relationships between them as shown in the figure.

But if you do this, you are making a mistake. In domain-driven design we let the *domain* drive the design, not a database schema.

It's better to work from the domain and to model it without respect to any particular storage implementation. After all, in a real-world, paper-based system, there is no database. The concept of a "database" is certainly not part of the ubiquitous language. The users do not care about how data is persisted.

In DDD terminology this is called *persistence ignorance*. It is an important principle because it forces you to focus on modeling the domain accurately, without worrying about the representation of the data in a database.

Why is this important? Well, if you design from the database point of view all the time, you often end up distorting the design to fit a database model.

As an example of the distortion that a database-driven model brings, we have already ignored the difference between an order and a quote in the diagram above. Sure, in the database we can have a flag to distinguish them, but the business rules and validation rules are different. For example, we might later learn that an `Order` must have a billing address but a `Quote` doesn't. This is hard to model with a foreign key. This subtlety has been lost in database design because the same foreign key does dual duty for both types of relationships.

Of course, the design can be corrected to deal with it, and in the chapter on persistence on page 239, we'll see how to persist a domain-driven design into a relational database. But for now we really want to concentrate on listening to the requirements without prejudice.

## Fighting the Impulse to Do Class-Driven Design

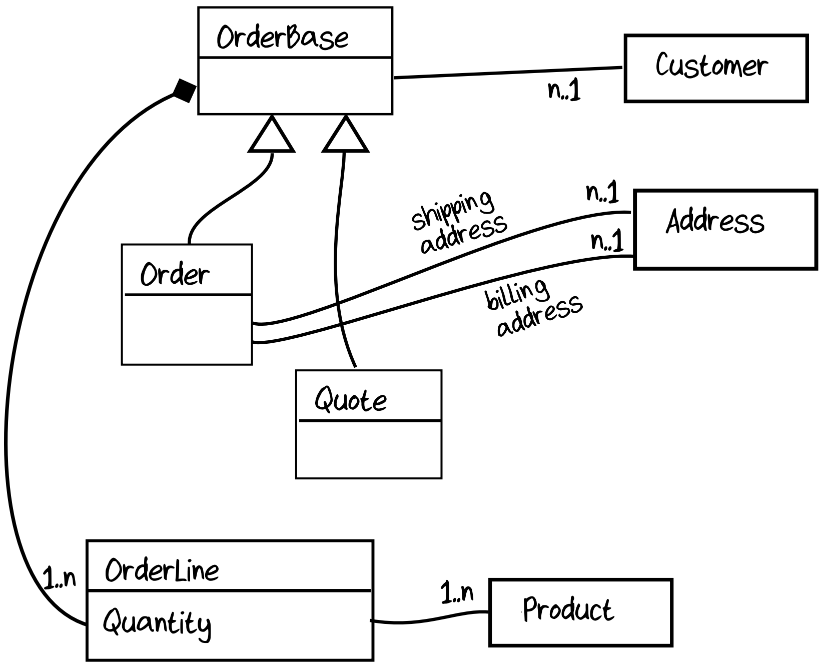
If you're an experienced object-oriented developer, then the idea of not being biased to a particular database model will be familiar, Indeed, object-oriented techniques such as dependency injection encourage you to separate the database implementation from the business logic.

But you, too, may end up introducing bias into the design if you think in terms of objects rather than the domain.

For example, as Ollie is talking, you may be creating classes in your head, like the figure on page 31.

Letting classes drive the design can be just as dangerous as letting a database drive the design—again, you're not really listening to the requirements.

In the preliminary design above we have separated orders and quotes, but we have introduced an artificial base class, `OrderBase`, that doesn't exist in the real world. This is a distortion of the domain. Try asking the domain expert what an `OrderBase` is!



The lesson here is that we should keep our minds open during requirements gathering and not impose our own technical ideas on the domain.

## Documenting the Domain

OK, we want to avoid biasing ourselves with technical implementations, but then how *should* we record these requirements?

We could use visual diagrams (such as UML), but these are often hard to work with and not detailed enough to capture some of the subtleties of the domain.

Later in this book we'll see how to create an accurate domain model in code, but for now, let's just create a simple text-based language that we can use to capture the domain model:

- For workflows, we'll document the inputs and outputs and then just use some simple pseudocode for the business logic.
- For data structures, we'll use AND to mean that both parts are required, such as in Name AND Address. And we'll use OR to mean that either part is required, such as in Email OR PhoneNumber.

Using this mini-language, then, we can document the Place Order workflow like this:

```
Bounded context: Order-Taking

Workflow: "Place order"
   triggered by:
      "Order form received" event (when Quote is not checked)
   primary input:
      An order form
   other input:
      Product catalog
   output events:
      "Order Placed" event
   side-effects:
      An acknowledgment is sent to the customer,
      along with the placed order
```

And we can document the data structures associated with the workflow like this:

```
bounded context: Order-Taking

data Order =
    CustomerInfo
    AND ShippingAddress
    AND BillingAddress
    AND list of OrderLines
    AND AmountToBill

data OrderLine =
    Product
    AND Quantity
    AND Price

data CustomerInfo = ???   // don't know yet
data BillingAddress = ??? // don't know yet
```

The Provide Quote workflow and its associated data structures can be documented in a similar way.

Note that we have not attempted to create a class hierarchy or database tables or anything else. We have just tried to capture the domain in a slightly structured way.

The advantage of this kind of text-based design is that it's not scary to non-programmers, which means it can be shown to the domain expert and worked on together.

The big question is whether can we make our code look as simple as this, too. In a following chapter, *Domain Modeling with Types*, we'll try to do just that.

## Diving Deeper into the Order-Taking Workflow

We've got the inputs and outputs documented, so let's move on to understanding the order-taking workflow in detail.

> You: "Ollie, could you go into detail on how you work with an order form?"
>
> Ollie: "When we get the mail in the morning, the first thing I do is sort it. Order forms are put on one pile, and other correspondence is put on another pile. Then, for each form, I look at whether the Quote box has been checked; if so, I put the form on the Quotes pile to be handled later."
>
> You: "Why is that?"
>
> Ollie: "Because orders are always more important. We make money on orders. We don't make money on quotes."

Ollie has mentioned something very important when gathering requirements. As developers, we tend to focus on technical issues and treat all requirements as equal. Businesses do not think that way. Making money (or saving money) is almost always the driver behind a development project. If you are in doubt as to what the most important priority is, follow the money! In this case, then, we need to design the system so that (money-making) orders are prioritized over quotes.

Moving on…

> You: "What's the first thing you do when processing an order form?"
>
> Ollie: "The first thing I do is check that the customer's name, email, shipping address, and billing address are valid."

After further discussion with Ollie, we learn that addresses are checked using a special application on Ollie's computer. Ollie types in the addresses, and the computer looks up whether they exist or not. It also puts them into a standard format that the delivery service likes.

We learned something new again. The workflow requires communication outside the context to some third-party address checking service. We missed that in the Event Storming, so we'll have to make a note of that.

If the name and addresses are not valid, Ollie marks the problems on the form with a red pen and puts it on the pile of invalid forms. Later on, Ollie will call the customer and ask to correct that information.

We are now aware of *three* piles: incoming order forms (from the mail), incoming quotes (to be processed later), and invalid order forms (also to be processed later).

Piles of paper are a very important part of most business processes. And let's reiterate that some piles are more important than other piles; we must not forget to capture this in our design. When we come to the implementation phase, a "pile of paper" corresponds nicely with a queue, but again we have to remind ourselves to stay away from technical details right now.

> Ollie: "After that, I check the product codes on the form. Sometimes they are obviously wrong."
>
> You: "How can you tell?"
>
> Ollie: "Because the codes have certain formats. The codes for widgets start with a *W* and then four digits. The codes for gizmos start with a *G* and then three digits."
>
> You: "Are there any other types of product codes? Or likely to be soon?"
>
> Ollie: "No. The product code formats haven't changed in years."
>
> You: "What about product codes that look right? Do you check that they are real products?"
>
> Ollie: "Yes. I look them up in my copy of the product catalog. If any codes are not there, I mark the form with the errors and put it in the pile of invalid orders."

Let's pause and look at what's going on with the product codes here:

- First, Ollie looks at the format of the code: does it start with a *W* or a *G*, and so on. In programming terms, this is a purely syntactic check. We don't need access to a product catalog to do that.
- Next, Ollie checks to see that the code exists in the product catalog. In Ollie's case, this involves looking something up in a book. In a software system, this would be a database lookup.

> You: "Here's a silly question. Let's say that someone on the product team could respond instantly to all your questions. Would you still need your own copy of the product catalog?"
>
> Ollie: "But what if they are busy? Or the phones were down? It's not really about speed, it's about control. I don't want my job to be interrupted because somebody else isn't available. If I have my own copy of the product catalog, I can process almost every order form without being dependent on the product team."

So this is really about dependency management, not performance. We discussed the importance of *autonomy* in relation to bounded contexts earlier (*Getting the Contexts Right,* on page 18). This may be important to model in the domain—or not—but either way you should be aware of the requirement for the departments to work independently.

> You: "OK, now say that all the product codes are good. What next?"
>
> Ollie: "I check the quantities."
>
> You: "Are the quantities integers or floats?"
>
> Ollie: "Float? Like in water?"

Ubiquitous language time! Pro tip: Domain experts do not use programming terms like "float."

> You: "What do you call those numbers then?"
>
> Ollie: "I call them 'order quantities,' duh!"

OK, we can see that `OrderQuantity` will need to be a word in the ubiquitous language, along with `ProductCode`, `AmountToBill`, and so on.

Let's try again:

> You: "Do the order quantities have decimals, or are they just whole numbers?"
>
> Ollie: "It depends."

"It depends." When you hear that, you know things are going to get complicated.

> You: "It depends on what?"
>
> Ollie: "It depends on what the product is. Widgets are sold by the unit, but gizmos are sold by the kilogram. If someone has asked for 1.5 widgets, then of course that's a mistake."

You scribble some notes down furiously.

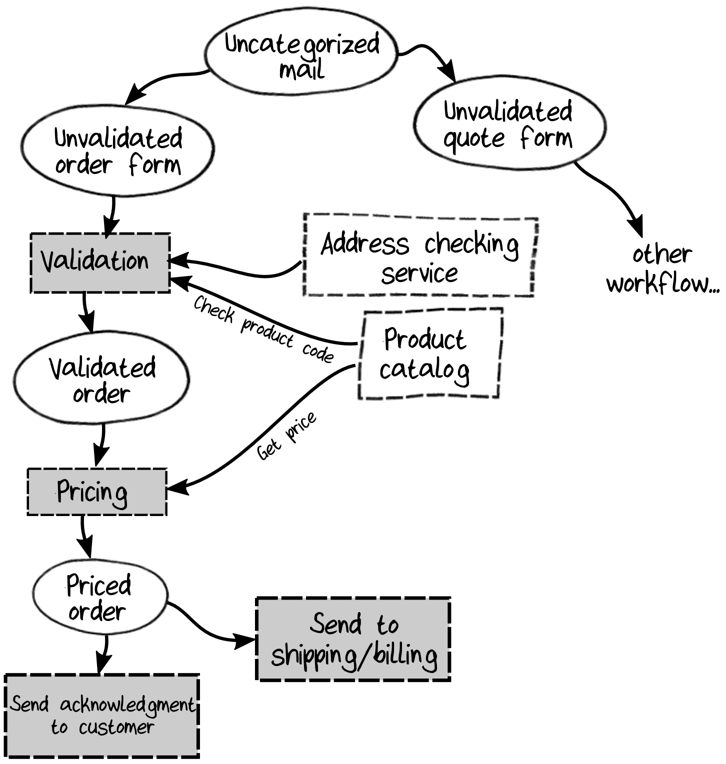
> You: "OK, say that all the product codes and order quantities are good. What next?"
>
> Ollie: "Next, I write in the prices for each line on the order and then sum them up to calculate the total amount to bill. Next, as I said earlier, I make two copies of the order form. I file the original and I put one copy in the shipping outbox and a second copy in the billing outbox. Finally, I scan the original, attach it to a standard acknowledgment letter, and email it back to the customer."
>
> You: "One last question. You have all these order forms lying around. Do you ever accidentally mix up ones you have processed with ones that are still unvalidated?"
>
> Ollie: "No. Every time I do something with them I mark them somehow. For example, when a form has been validated, I put a mark up here in the corner, so I know I've done that. I can tell when I've calculated the prices because the "total" box is filled out. Doing this means I can always tell order forms at different stages apart."

This is a good point to stop and digest what we've learned.

## Representing Complexity in Our Domain Model

As we have drilled down into this one workflow, the domain model has become a lot more complicated. That's good. Better to spend time on understanding complexity now rather than later, when we are in the middle of coding. "A few weeks of programming can save you hours of planning," as they say.

Here's a diagram of the workflow now:



But this diagram doesn't reflect everything that we've learned. Let's see if we can do better and capture all this new information in our text-based language.

### Representing Constraints

We'll start with the most primitive values first: the product codes and quantities, which we've learned are not just simple strings and integers but are constrained in various ways.

```
context: Order-Taking

data WidgetCode = string starting with "W" then 4 digits
data GizmoCode = string starting with "G" then 3 digits
data ProductCode = WidgetCode OR GizmoCode
```

In the fragment above, the words that Ollie used (such as WidgetCode) have also been used for the design, and we are treating them as part of the Ubiquitous Language. Furthermore, we have documented the constraints on WidgetCode and GizmoCode and then defined a ProductCode as a choice between those two types.

But isn't that being too strict? What happens if a new type of product needs to be handled? This is a problem we frequently run into. If we are too strict, we make things harder to change. But if we have too much freedom, we don't have a design at all.

The right answer depends on the context, as always. Generally though, it's important to capture the design from the domain expert's point of view. Checking the different kinds of codes is an important part of the validation process, and so it should be reflected in the design of the domain, which aims to be self-documenting. And if we didn't document the different kinds of product codes here, as part of the model, we'd have to document them somewhere else anyway.

Also, if the requirements *do* change, our model is very easy to change; adding a new kind of product code would only require an extra line.

Finally, remember that just because the design is strict doesn't mean that the implementation has to be strict. For example, an automated version of the validation process might just flag a strange code for human approval, rather than rejecting the whole order outright.

Now, what about documenting the requirements for the quantities? Here's the proposed design:

```
data OrderQuantity = UnitQuantity OR KilogramQuantity

data UnitQuantity = integer between 1 and ?
data KilogramQuantity = decimal between ? and ?
```

Just as we did with product codes, we'll define OrderQuantity as a choice—in this case between UnitQuantity and KilogramQuantity.

Writing this down, though, we realize that we don't have upper bounds for UnitQuantity and KilogramQuantity. Surely UnitQuantity can't be allowed to be in the billions?

Let's check with the domain expert. Ollie gives us the limits we need:

- The largest number of units allowed for an order quantity is 1000.
- The lowest weight is 0.05 kg and the highest is 100 kg.

These kinds of constraints are important to capture. We never want a situation in production where the units are accidentally negative, or the weight is hundreds of kilotons. Here is the updated spec, with these constraints documented:

```
data UnitQuantity = integer between 1 and 1000
data KilogramQuantity = decimal between 0.05 and 100.00
```

## Representing the Life Cycle of an Order

Now let's move on to the Order. In our earlier design sketch, we had a simple definition for Order:

```
data Order =
    CustomerInfo
    AND ShippingAddress
    AND BillingAddress
    AND list of OrderLines
    AND AmountToBill
```

But now it's clear that this design is too simplistic and doesn't capture how Ollie thinks of orders. In Ollie's mental model, orders have a life cycle. They start off as unvalidated (straight from the mail), then they get "validated," and then they get "priced."

In the beginning, an order doesn't have a price, but by the end it does. The simple Order definition above erases that distinction.

With the paper forms, Ollie distinguishes between these phases by putting marks on the order after each phase, so an unvalidated order is immediately distinguishable from a validated one, and a validated one from a priced one.

We need to capture these same phases in our domain model, not just for documentation but to make it clear that (for example) an unpriced order should not be sent to the shipping department.

The easiest way to do that is by creating new names for each phase: UnvalidatedOrder, ValidatedOrder, and so on. It does mean that the design becomes longer and more tedious to write out, but the advantage is that everything is crystal clear.

Let's start with the initial unvalidated orders and quotes that arrive. We can document them like this:

```
data UnvalidatedOrder =
    UnvalidatedCustomerInfo
    AND UnvalidatedShippingAddress
    AND UnvalidatedBillingAddress
    AND list of UnvalidatedOrderLine
```

```
data UnvalidatedOrderLine =
    UnvalidatedProductCode
    AND UnvalidatedOrderQuantity
```

This documentation makes it explicit that at the beginning of the workflow, the CustomerInfo is not yet validated, the ShippingAddress is not yet validated, and so on.

The next stage is when the order has been validated. We can document it like this:

```
data ValidatedOrder =
    ValidatedCustomerInfo
    AND ValidatedShippingAddress
    AND ValidatedBillingAddress
    AND list of ValidatedOrderLine

data ValidatedOrderLine =
    ValidatedProductCode
    AND ValidatedOrderQuantity
```

This shows that all the components have now been checked and are valid.

The next stage is to price the order. A Priced Order is just like a validated order except for the following:

- Each line now has a price associated with it. That is, a PricedOrderLine is a ValidatedOrderLine plus a LinePrice.
- The order as a whole has an AmountToBill associated with it, calculated as the sum of the line prices.

Here's the model for this:

```
data PricedOrder =
    ValidatedCustomerInfo
    AND ValidatedShippingAddress
    AND ValidatedBillingAddress
    AND list of PricedOrderLine  // different from ValidatedOrderLine
    AND AmountToBill             // new

data PricedOrderLine =
    ValidatedOrderLine
    AND LinePrice                // new
```

The final stage is to create the order acknowledgment.

```
data PlacedOrderAcknowledgment =
    PricedOrder
    AND AcknowledgmentLetter
```

You can see now that we've captured quite a lot of the business logic in this design already—rules such as these:

- An unvalidated order does not have a price.
- All the lines in a validated order must be validated, not just some of them.

The model is a lot more complicated than we originally thought. But we are just reflecting the way that the business works. If our model wasn't this complicated, we wouldn't be capturing the requirements properly.

Now if we can preserve these distinctions in our code as well, then our code will reflect the domain accurately and we will have a proper "domain-driven" design.

## Fleshing out the Steps in the Workflow

It should be apparent that the workflow can be broken down into smaller steps: validation, pricing, and so on. Let's apply the same input/output approach to each of these steps.

First, the output of the overall workflow is a little more complicated than we thought earlier. Originally the only output was a "Order placed" event, but now the possible outcomes for the workflow are as follows:

- We send a "Order placed" event to shipping/billing, OR
- We add the order form to the invalid order pile and skip the rest of the steps.

Let's document the whole workflow in pseudocode, with steps like ValidateOrder broken out into separate substeps:

```
workflow "Place Order" =
    input: OrderForm
    output:
       OrderPlaced event (put on a pile to send to other teams)
       OR InvalidOrder (put on appropriate pile)

    // step 1
    do ValidateOrder
    If order is invalid then:
        add InvalidOrder to pile
        stop

    // step 2
    do PriceOrder

    // step 3
    do SendAcknowledgmentToCustomer

    // step 4
    return OrderPlaced event (if no errors)
```

With the overall flow documented, we can now add the extra details for each substep.

For example, the substep that validates the form takes an UnvalidatedOrder as input, and its output is either a ValidatedOrder or a ValidationError. We will also document the dependencies for the substep: it needs input from the product catalog (we'll call this the CheckProductCodeExists dependency) and the external address checking service (the CheckAddressExists dependency).

```
substep "ValidateOrder" =
    input: UnvalidatedOrder
    output: ValidatedOrder OR ValidationError
    dependencies: CheckProductCodeExists, CheckAddressExists

    validate the customer name
    check that the shipping and billing address exist
    for each line:
        check product code syntax
        check that product code exists in ProductCatalog

    if everything is OK, then:
        return ValidatedOrder
    else:
        return ValidationError
```

The substep that calculates the prices takes a ValidatedOrder as input and has a dependency on the product catalog (which we'll call GetProductPrice). The output is a PricedOrder.

```
substep "PriceOrder" =
    input: ValidatedOrder
    output: PricedOrder
    dependencies: GetProductPrice

    for each line:
        get the price for the product
        set the price for the line
    set the amount to bill ( = sum of the line prices)
```

Finally, the last substep takes a PricedOrder as input and then creates and sends the acknowledgment.

```
substep "SendAcknowledgmentToCustomer" =
    input: PricedOrder
    output: None

    create acknowledgment letter and send it
    and the priced order to the customer
```

This documentation of the requirements is looking a lot more like code now, but it can still be read and checked by a domain expert.

## Wrapping Up

We'll stop gathering requirements now, as we'll have plenty to work with when we move to the modeling phase in the second part of this book. But first let's review what we've learned in this chapter.

We saw that it's important not to dive into implementation details while doing design: DDD is neither database-driven nor class-driven. Instead, we focused on capturing the domain without assumptions and without assuming any particular way of coding.

And we saw that listening to the domain expert carefully reveals a lot of complexity, even in a relatively simple system like this. For example, we originally thought that there would be a single "Order," but more investigation uncovered many variants of an order throughout its life cycle, each with slightly different data and behavior.

### What's Next

We'll be looking shortly at how we can model this order-taking workflow using the F# type system. But before we do that, let's step back and look at the big picture again and discuss how to translate a complete system into a software architecture. That will be the topic of the next chapter.