CHAPTER 5

Domain Modeling with Types

In the first chapter, when we were talking about the importance of a shared mental model, we emphasized that the code must also reflect this shared model and that a developer should not have to do lossy translations between the domain model and the source code. Ideally, we would like the source code to also act as documentation, which means that the domain expert and other non-developers should be able to review the code and check on the design.

Is that a realistic goal? Can we use the source code directly like this and avoid the need for UML diagrams and the like?

The answer is yes. In this chapter you'll learn how to use the F# type system to capture the domain model accurately enough for code but also in a way that can be read and understood by domain experts and other non-developers. We'll see that types can replace most documentation, and that ability has a powerful benefit: the implementation can never get out of sync with the design because the design is represented in code itself.

Reviewing the Domain Model

Let's review the domain model that we created previously on page 36:

```
context: Order-Taking

// ----------------------
// Simple types
// ----------------------

// Product codes
data ProductCode = WidgetCode OR GizmoCode
data WidgetCode = string starting with "W" then 4 digits
data GizmoCode = ...
```

```
// Order Quantity
data OrderQuantity = UnitQuantity OR KilogramQuantity
data UnitQuantity = ...
data KilogramQuantity = ...

// ----------------------
// Order life cycle
// ----------------------

// ----- unvalidated state -----
data UnvalidatedOrder =
    UnvalidatedCustomerInfo
    AND UnvalidatedShippingAddress
    AND UnvalidatedBillingAddress
    AND list of UnvalidatedOrderLine

data UnvalidatedOrderLine =
    UnvalidatedProductCode
    AND UnvalidatedOrderQuantity

// ----- validated state -----
data ValidatedOrder = ...
data ValidatedOrderLine =  ...

// ----- priced state -----
data PricedOrder = ...
data PricedOrderLine = ...

// ----- output events -----
data OrderAcknowledgmentSent = ...
data OrderPlaced = ...
data BillableOrderPlaced = ...

// ----------------------
// Workflows
// ----------------------

workflow "Place Order" =
    input: UnvalidatedOrder
    output (on success):
        OrderAcknowledgmentSent
        AND OrderPlaced (to send to shipping)
        AND BillableOrderPlaced (to send to billing)
    output (on error):
        InvalidOrder

// etc
```

The goal of this chapter is to turn this model into code.

Seeing Patterns in a Domain Model

Although each domain model is different, many patterns occur repeatedly.

Let's look at some of the patterns of a typical domain and see how we can relate components of our model to them.

- *Simple values*. These are the basic building blocks represented by primitive types such as strings and integers. But note that they are *not* actually strings or integers. A domain expert does not think in terms of int and string, but instead thinks in terms of OrderId and ProductCode—concepts that are part of the ubiquitous language.
- *Combinations of values with* AND. These are groups of closely linked data. In a paper-based world, these are typically documents or subcomponents of a document: names, addresses, orders, and so forth.
- *Choices with* OR. We have things that represent a choice in our domain: an Order *or* a Quote, a UnitQuantity *or* a KilogramQuantity.
- *Workflows*. Finally, we have business processes that have inputs and outputs.

In the next few sections, we'll look at how we can represent these different patterns using F# types.

Modeling Simple Values

Let's first look at the building blocks of a domain: simple values.

As we found out when we gathered the requirements on page 33, a domain expert does not generally think in terms of int and string but instead in terms of domain concepts such as OrderId and ProductCode. Furthermore, it's important that OrderIds and ProductCodes don't get mixed up. Just because they're both represented by ints, say, doesn't mean that they are interchangeable. So to make it clear that these types are distinct, we'll create a "wrapper type"—a type that wraps the primitive representation.

As we mentioned earlier, the easiest way to create a wrapper type in F# is to create a "single-case" union type, a choice type with only one choice.

Here's an example:

```
type CustomerId =
  | CustomerId of int
```

Since there's only one case, we invariably write the whole type definition on one line, like this:

```
type CustomerId = CustomerId of int
```

We'll call these kinds of wrapper types "simple types" to distinguish them both from compound types (such as records) and the raw primitive types (such as string and int) that they contain.

In our domain, the simple types would be modeled this way:

```
type WidgetCode = WidgetCode of string
type UnitQuantity = UnitQuantity of int
type KilogramQuantity = KilogramQuantity of decimal
```

The definition of a single case union has two parts: the name of the type and the "case" label:

```
type CustomerId = CustomerId of int
//   ^type name   ^case label
```

As you can see from the examples above, the label of the (single) case is typically the same as the name of the type. This means that when using the type, you can also use the same name for constructing and deconstructing it, as we'll see next.

Working with Single Case Unions

To create a value of a single case union, we use the case name as a constructor function. That is, we've defined a simple type like this:

```
type CustomerId = CustomerId of int
//                ^this case name will be the constructor function
```

Now we can create it by using the case name as a constructor function:

```
let customerId = CustomerId 42
//               ^this is a function with an int parameter
```

Creating simple types like this ensures that we can't confuse different types by accident. For example, if we create a CustomerId and an OrderId and try to compare them, we get a compiler error:

```
// define some types
type CustomerId = CustomerId of int
type OrderId = OrderId of int

// define some values
let customerId = CustomerId 42
let orderId = OrderId 42

// try to compare them -- compiler error!
printfn "%b" (orderId = customerId)
//                      ^ This expression was expected to
//                      have type 'OrderId'
```

Or if we have defined a function that takes a CustomerId as input, then trying to pass it an OrderId is another compiler error:

```
// define a function using a CustomerId
let processCustomerId (id:CustomerId) = ...

// call it with an OrderId -- compiler error!
processCustomerId orderId
//                ^ This expression was expected to
//                have type 'CustomerId' but here has
//                type 'OrderId'
```

To deconstruct or unwrap a single case union, we can pattern-match using the case label:

```
// construct
let customerId = CustomerId 42

// deconstruct
let (CustomerId innerValue) = customerId
//              ^ innerValue is set to 42

printfn "%i" innerValue  // prints "42"
```

It's very common to deconstruct directly in the parameter of a function definition. When we do this, we not only can access the inner value immediately but the F# compiler will also infer the correct type for us. For example, in the code below, the compiler infers the input parameter is a CustomerId:

```
// deconstruct
let processCustomerId (CustomerId innerValue) =
  printfn "innerValue is %i" innerValue

// function signature
// val processCustomerId: CustomerId -> unit
```

Constrained Values

Almost always, the simple types are constrained in some way, such as having to be in a certain range or match a certain pattern. It's very rare to have an unbounded integer or string in a real-world domain.

We'll discuss how to enforce these constraints in the next chapter (*The Integrity of Simple Values*, on page 104).

Avoiding Performance Issues with Simple Types

Wrapping primitive types into simple types is a great way to ensure type-safety and prevent many errors at compile time. However, it does come at a cost in memory usage and efficiency. For typical business applications a small decrease in performance shouldn't be a problem, but for domains that require

high performance, such as scientific or real-time domains, you might want to be more careful. For example, looping over a large array of UnitQuantity values will be slower than looping over an array of raw ints.

But there are a couple of ways you can have your cake and eat it too.

First, you can use type aliases instead of simple types to document the domain. This has no overhead, but it does mean a loss of type-safety.

```
type UnitQuantity = int
```

Next, as of F# 4.1, you can use a value type (a struct) rather than a reference type. You'll still have overhead from the wrapper, but when you store them in arrays the memory usage will be contiguous and thus more cache-friendly.

```
[<Struct>]
type UnitQuantity = UnitQuantity of int
```

Finally, if you are working with large arrays, consider defining the entire collection of primitive values as a single type rather than having a collection of simple types:

```
type UnitQuantities = UnitQuantities of int[]
```

This will give you the best of both worlds. You can work efficiently with the raw data (such as for matrix multiplication) while preserving type-safety at a high level. Extending this approach further leads you to data-oriented design,[1] as used in modern game development.

You might even find that there is a word in the ubiquitous language for these kinds of collections that are treated as a unit, such as "DataSample" or "Measurements." If so, use it!

As always, performance is a complex topic and depends on your specific code and environment. It's generally best to model your domain in the most straightforward way first and only then work on tuning and optimization.

Modeling Complex Data

When we documented our domain on page 31, we used *AND* and *OR* to represent more complex models. In *Understanding Types*, we learned about F#'s algebraic type system and saw that it also used *AND* and *OR* to create complex types from simple ones.

Let's now take the obvious step and use the algebraic type system to model our domain.

1. https://en.wikipedia.org/wiki/Data-oriented_design

Modeling with Record Types

In our domain, we saw that many data structures were built from *AND* relationships. For example, our original, simple Order was defined like this:

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

This translates directly to an F# record structure, like this:

```
type Order = {
  CustomerInfo : CustomerInfo
  ShippingAddress : ShippingAddress
  BillingAddress : BillingAddress
  OrderLines : OrderLine list
  AmountToBill : ...
  }
```

We have given each field a name ("CustomerInfo," "ShippingAddress") and a type (CustomerInfo, ShippingAddress).

Doing this shows a lot of still-unanswered questions about the domain—we don't know what these types actually are right now. Is ShippingAddress the same type as BillingAddress? What type should we use to represent "AmountToBill"?

Ideally, we can ask our domain experts to help with this. For example, if your experts talk about billing addresses and shipping addresses as different things, it's better to keep these logically separate, even if they have the same structure. They may evolve in different directions as your domain understanding improves or as requirements change.

Modeling Unknown Types

During the early stages of the design process, you often won't have definitive answers to some modeling questions. For example, you'll know the names of types that you need to model, thanks to the ubiquitous language, but not their internal structure.

This isn't a problem—you can represent types of unknown structure with best guesses, or alternatively you can model them as a type that's *explicitly* undefined, one that acts as a placeholder, until you have a better understanding later in the design process.

If you want to represent an undefined type in F#, you can use the exception type `exn` and alias it to `Undefined`:

```
type Undefined = exn
```

You can then use the `Undefined` alias in your design model, like this:

```
type CustomerInfo = Undefined
type ShippingAddress = Undefined
type BillingAddress = Undefined
type OrderLine = Undefined
type BillingAmount = Undefined

type Order = {
  CustomerInfo : CustomerInfo
  ShippingAddress : ShippingAddress
  BillingAddress : BillingAddress
  OrderLines : OrderLine list
  AmountToBill : BillingAmount
  }
```

This approach means that you can keep modeling the domain with types and compile the code. But when you try to write the functions that process the types, you will be forced to replace `Undefined` with something a bit better.

Modeling with Choice Types

In our domain, we also saw many things that were choices between other things, such as these:

```
data ProductCode =
    WidgetCode
    OR GizmoCode

data OrderQuantity =
    UnitQuantity
    OR KilogramQuantity
```

How can we represent these choices with the F# type system? With choice types, obviously!

```
type ProductCode =
  | Widget of WidgetCode
  | Gizmo of GizmoCode

type OrderQuantity =
  | Unit of UnitQuantity
  | Kilogram of KilogramQuantity
```

Again, for each case we need to create two parts: the "tag" or case label (before the "of") and the type of the data that is associated with that case. The

example above shows that the case label (such as Widget) doesn't have to be the same as the name of the type (WidgetCode) associated with it.

Modeling Workflows with Functions

We've now got a way to model all the data structures—the "nouns" of the ubiquitous language. But what about the "verbs," the business processes? In this book, we will model workflows and other processes as function types. For example, if we have a workflow step that validates an order form, we might document it like this:

```
type ValidateOrder = UnvalidatedOrder-> ValidatedOrder
```

It's clear from this code that the ValidateOrder process transforms an unvalidated order into a validated one.

Working with Complex Inputs and Outputs

Every function has only one input and one output, but some workflows might have multiple inputs and outputs. How can we model that? We'll start with the outputs. If a workflow has an outputA *and* an outputB, then we can create a record type to store them both. We saw this with the order-placing workflow: the output needs to be three different events, so let's create a compound type to store them as one record:

```
type PlaceOrderEvents = {
  AcknowledgmentSent : AcknowledgmentSent
  OrderPlaced : OrderPlaced
  BillableOrderPlaced : BillableOrderPlaced
  }
```

Using this approach, the order-placing workflow can be written as a function type, starting with the raw UnvalidatedOrder as input and returning the PlaceOrderEvents record:

```
type PlaceOrder = UnvalidatedOrder -> PlaceOrderEvents
```

On the other hand, if a workflow has an outputA *or* an outputB, then we can create a choice type to store them both. For example, we briefly talked about categorizing the inbound mail as quotes or orders on page 33. That process had at least two different choices for outputs:

```
workflow "Categorize Inbound Mail" =
    input: Envelope contents
    output:
        QuoteForm (put on appropriate pile)
        OR OrderForm (put on appropriate pile)
        OR ...
```

It's easy to model this workflow: just create a new type, say CategorizedMail, to represent the choices, and then have CategorizeInboundMail return that type. Our model might then look like this:

```
type EnvelopeContents = EnvelopeContents of string
type CategorizedMail =
  | Quote of QuoteForm
  | Order of OrderForm
  // etc

type CategorizeInboundMail = EnvelopeContents -> CategorizedMail
```

Now let's look at modeling inputs. If a workflow has a choice of different inputs (*OR*), then we can create a choice type. But if a process has multiple inputs that are all required (*AND*), such as "Calculate Prices" (below), we can choose between two possible approaches.

```
"Calculate Prices" =
    input: OrderForm, ProductCatalog
    output: PricedOrder
```

The first and simplest approach is just to pass each input as a separate parameter, like this:

```
type CalculatePrices = OrderForm -> ProductCatalog -> PricedOrder
```

Alternatively, we could create a new record type to contain them both, such as this CalculatePricesInput type:

```
type CalculatePricesInput = {
  OrderForm : OrderForm
  ProductCatalog : ProductCatalog
  }
```

And now the function looks like this:

```
type CalculatePrices = CalculatePricesInput -> PricedOrder
```

Which approach is better? In the cases above, where the ProductCatalog is a dependency rather than a "real" input, we want to use the separate parameter approach. This lets us use the functional equivalent of dependency injection. We'll discuss this in detail in *Injecting Dependencies*, on page 180, when we implement the order-processing pipeline.

On the other hand, if both inputs are always required and are strongly connected with each other, then a record type will make that clear. (In some situations, you can use tuples as an alternative to simple record types, but it's generally better to use a named type.)

Documenting Effects in the Function Signature

We just saw that the ValidateOrder process could be written like this:

```
type ValidateOrder = UnvalidatedOrder -> ValidatedOrder
```

But that assumes that the validation always works and a ValidatedOrder is always returned. In practice, of course, this would not be true, so it would better to indicate this situation by returning a Result type (introduced on page 70) in the function signature:

```
type ValidateOrder =
  UnvalidatedOrder -> Result<ValidatedOrder,ValidationError list>

and ValidationError = {
  FieldName : string
  ErrorDescription : string
  }
```

This signature shows us that the input is an UnvalidatedOrder and, if successful, the output is a ValidatedOrder. But if validation failed, the result is a list of ValidationError, which in turn contains a description of the error and which field it applies to.

Functional programming people use the term *effects* to describe things that a function does in addition to its primary output. By using Result here, we've now documented that ValidateOrder might have "error effects." This makes it clear in the type signature that we can't assume the function will always succeed and that we should be prepared to handle errors.

Similarly, we might want to document that a process is asynchronous—it will not return immediately. How can we do that? With another type of course!

In F#, we use the Async type to show that a function will have "asynchronous effects." So if ValidateOrder had async effects as well as error effects, then we would write the function type like this:

```
type ValidateOrder =
  UnvalidatedOrder -> Async<Result<ValidatedOrder,ValidationError list>>
```

This type signature now documents (a) when we attempt to fetch the contents of the return value, the code won't return immediately and (b) when it does return, the result might be an error.

Listing all the effects explicitly like this is useful, but it does make the type signature ugly and complicated, so we would typically create a type alias for this to make it look nicer.

```
type ValidationResponse<'a> = Async<Result<'a,ValidationError list>>
```

Then the function could be documented like this:

```
type ValidateOrder =
  UnvalidatedOrder -> ValidationResponse<ValidatedOrder>
```

A Question of Identity: Value Objects

We've now got a basic understanding of how to model the domain types and workflows, so let's move on and look at an important way of classifying data types based on whether they have a persistent identity or not.

In DDD terminology, objects with a persistent identity are called *Entities* and objects without a persistent identity are called *Value Objects*. Let's start by discussing Value Objects first.

In many cases, the data objects we're dealing with have no identity—they're interchangeable. For example, one instance of a WidgetCode with value "W1234" is the same as any other WidgetCode with value "W1234." We don't need to keep track of which one is which—they're equal to each other.

In F# we might demonstrate this as follows:

```
let widgetCode1 = WidgetCode "W1234"
let widgetCode2 = WidgetCode "W1234"
printfn "%b" (widgetCode1 = widgetCode2)  // prints "true"
```

The concept of "values without identity" shows up frequently in a domain model, and for complex types as well as simple types. For example, a PersonalName record type might have two fields—FirstName and LastName— so it's more complex than a simple string; but it's also a *Value Object*, because two personal names with the same fields are interchangeable. We can see that with the following F# code:

```
let name1 = {FirstName="Alex"; LastName="Adams"}
let name2 = {FirstName="Alex"; LastName="Adams"}
printfn "%b" (name1 = name2)  // prints "true"
```

An "address" type is also a Value Object. If two values have the same street address, city, and zip code, they are the same address:

```
let address1 = {StreetAddress="123 Main St"; City="New York"; Zip="90001"}
let address2 = {StreetAddress="123 Main St"; City="New York"; Zip="90001"}
printfn "%b" (address1 = address2)  // prints "true"
```

You can tell that these are Value Objects in the domain because when discussing them, you would say something like, "Chris has the same name as me." That is, even though Chris and I are different people, our *names* are the same. They don't have a unique identity. Similarly, "Pat has the same postal

address as me" means that my address and Pat's address have the same content and are thus equal.

Implementing Equality for Value Objects

When we model the domain using the F# algebraic type system, the types we create will implement this kind of field-based equality testing by default. We don't need to write any special equality code ourselves, which is nice.

To be precise, two record values (of the same type) are equal in F# if all their fields are equal, and two choice types are equal if they have the same choice case and the data associated with that case is also equal. This is called *structural equality*.

A Question of Identity: Entities

However, we often model things that, in the real world, *do* have a unique identity, even as their components change. For example, even if I change my name or my address, I am still the same person.

In DDD terminology, we call such things *Entities*.

In a business context, Entities are often a document of some kind: orders, quotes, invoices, customer profiles, product sheets, and so on. They have a *life cycle* and are transformed from one state to another by various business processes.

The distinction between "Value Object" and "Entity" is context-dependent. For example, consider the life cycle of a cell phone. During manufacturing, each phone is given a unique serial number—a unique identity—so in that context, the phone would be modeled as an Entity. When they're being sold, however, the serial number isn't relevant—all phones with the same specs are interchangeable—and they can be modeled as Value Objects. But once a particular phone is sold to a particular customer, identity becomes relevant again and it should be modeled as an Entity: the customer thinks of it as the same phone even after replacing the screen or battery.

Identifiers for Entities

Entities need to have a stable identity despite any changes. Therefore, when modeling them we need to give them a unique identifier or key, such as an "Order ID" or "Customer ID."

For example, the `Contact` type below has a `ContactId` that stays the same even if the `PhoneNumber` or `EmailAddress` fields change:

```
type ContactId = ContactId of int

type Contact = {
  ContactId : ContactId
  PhoneNumber : ...
  EmailAddress: ...
  }
```

Where do these identifiers come from? Sometimes the identifier is provided by the real-world domain itself—paper orders and invoices have always had some kind of reference written on them—but sometimes we'll need to create an artificial identifier ourselves using techniques such as UUIDs, an auto-incrementing database table, or an ID-generating service. This is a complex topic, so in this book we'll just assume that any identifiers have been provided to us by the client.

Adding Identifiers to Data Definitions

Given that we have identified a domain object as an Entity, how do we add an identifier to its definition?

Adding an identifier to a record type is straightforward—just add a field—but what about adding an identifier to a choice type? Should we put the identifier *inside* (associated with each case) or *outside* (not associated with any of the cases)?

For example, say that we have two choices for an 'Invoice': paid and unpaid. If we model it using the "outside" approach, we'll have a record containing the 'InvoiceId', and then within that record we'll have a choice type 'InvoiceInfo' that has information for each type of invoice. The code will look something like this:

```
// Info for the unpaid case (without id)
type UnpaidInvoiceInfo = ...

// Info for the paid case (without id)
type PaidInvoiceInfo = ...

// Combined information (without id)
type InvoiceInfo =
  | Unpaid of UnpaidInvoiceInfo
  | Paid of PaidInvoiceInfo

// Id for invoice
type InvoiceId = ...

// Top level invoice type
type Invoice = {
  InvoiceId : InvoiceId // "outside" the two child cases
  InvoiceInfo : InvoiceInfo
  }
```

The problem with this approach is that it's hard to work with the data for one case easily because it's spread between different components.

In practice, it's more common to store the ID using the "inside" approach, where each case has a copy of the identifier. Applied to our example, we would create two separate types, one for each case (UnpaidInvoice and PaidInvoice), both of which have their own InvoiceId, and then a top-level Invoice type, which is a choice between them. The code will look something like this:

```
type UnpaidInvoice = {
  InvoiceId : InvoiceId // id stored "inside"
  // and other info for the unpaid case
  }

type PaidInvoice = {
  InvoiceId : InvoiceId // id stored "inside"
  // and other info for the paid case
  }

// top level invoice type
type Invoice =
  | Unpaid of UnpaidInvoice
  | Paid of PaidInvoice
```

The benefit of this approach is that now, when we do our pattern matching, we have all the data accessible in one place, including the ID:

```
let invoice = Paid {InvoiceId = ...}

match invoice with
  | Unpaid unpaidInvoice ->
    printfn "The unpaid invoiceId is %A" unpaidInvoice.InvoiceId
  | Paid paidInvoice ->
    printfn "The paid invoiceId is %A" paidInvoice.InvoiceId
```

Implementing Equality for Entities

We saw earlier that, by default, equality testing in F# uses *all* the fields of a record. But when we compare Entities we want to use only one field, the identifier. That means that in order to model Entities correctly in F#, we must change the default behavior.

One way of doing this is to override the equality test so that only the identifier is used. To change the default we have to do the following:

1. Override the Equals method
2. Override the GetHashCode method
3. Add the CustomEquality and NoComparison attributes to the type to tell the compiler that we want to change the default behavior

When we do all this to the Contact type, we get this result:

```
[<CustomEquality; NoComparison>]
type Contact = {
  ContactId : ContactId
  PhoneNumber : PhoneNumber
  EmailAddress: EmailAddress
  }
  with
  override this.Equals(obj) =
    match obj with
    | :? Contact as c -> this.ContactId = c.ContactId
    | _ -> false
  override this.GetHashCode() =
    hash this.ContactId
```

This is a new kind of syntax we haven't seen yet: F#'s object-oriented syntax. We are only using it here to demonstrate equality overriding, but object-oriented F# is out of scope, so we won't use it elsewhere in the book.

With the type defined, we can create one contact:

```
let contactId = ContactId 1

let contact1 = {
  ContactId = contactId
  PhoneNumber = PhoneNumber "123-456-7890"
  EmailAddress = EmailAddress "bob@example.com"
  }
```

And create a different contact with the same ContactId:

```
// same contact, different email address
let contact2 = {
  ContactId = contactId
  PhoneNumber = PhoneNumber "123-456-7890"
  EmailAddress = EmailAddress "robert@example.com"
  }
```

Finally, when we compare them using =, the result is true:

```
// true even though the email addresses are different
printfn "%b" (contact1 = contact2)
```

This is a common approach in object-oriented designs, but by changing the default equality behavior silently it can trip you up on occasion. Therefore, an (often preferable) alternative is to disallow equality testing on the object altogether by adding a NoEquality type annotation like this:

```
[<NoEquality; NoComparison>]
type Contact = {
  ContactId : ContactId
  PhoneNumber : PhoneNumber
  EmailAddress: EmailAddress
  }
```

Now when we attempt to compare values with this annotation, we get a compiler error:

```
// compiler error!
printfn "%b" (contact1 = contact2)
//            ^ the Contact type does not
//              support equality
```

Of course we can still compare the ContactId fields directly, like this:

```
// no compiler error
printfn "%b" (contact1.ContactId = contact2.ContactId) // true
```

The benefit of the "NoEquality" approach is that it removes any ambiguity about what equality means at the object level and forces us to be explicit.

Finally, in some situations, you might have multiple fields that are used for testing equality. In this case, you can easily expose a synthetic Key property that combines them:

```
[<NoEquality;NoComparison>]
type OrderLine = {
  OrderId : OrderId
  ProductId : ProductId
  Qty : int
  }
  with
  member this.Key =
    (this.OrderId,this.ProductId)
```

And then, when you need to do a comparison, you can use the Key field, like this:

```
printfn "%b" (line1.Key = line2.Key)
```

Immutability and Identity

As we saw in *Understanding Types*, values in functional programming languages like F# are immutable by default, which means that none of the objects defined so far can be changed after being initialized.

How does this affect our design?

- For *Value Objects*, immutability is required. Think of how we use them in common speech: if we change any part of a personal name, say, we call it a *new*, distinct name, not the same name with different data.
- For *Entities*, it's a different matter. We expect the data associated with Entities to change over time; that's the whole point of having a constant identifier. So how can immutable data structures be made to work this way? The answer is that we make a *copy* of the Entity with the changed data while preserving the identity. All this copying might seem like a lot of extra work but isn't an issue in practice. In fact, throughout this book we will be using immutable data everywhere, and you will see that immutability is rarely a problem.

Here's an example of how an Entity can be updated in F#. First, we'll start with an initial value:

```
let initialPerson = {PersonId=PersonId 42; Name="Joseph"}
```

To make a copy of the record while changing only some fields, F# uses the `with` keyword, like this:

```
let updatedPerson = {initialPerson with Name="Joe"}
```

After this copy, the `updatedPerson` value has a different `Name` but the same `PersonId` as the `initialPerson` value.

A benefit of using immutable data structures is that any changes have to be made explicit in the type signature. For example, if we want to write a function that changes the `Name` field in a `Person`, we can't use a function with a signature, like this:

```
type UpdateName = Person -> Name -> unit
```

That function has no output, which implies that nothing changed (or that the `Person` was mutated as a side effect). Instead, our function must have a signature with the `Person` type as the output, like this:

```
type UpdateName = Person -> Name -> Person
```

This clearly indicates that, given a `Person` and a `Name`, some kind of variant of the original `Person` is being returned.

Aggregates

Let's take a closer look at two data types that are especially relevant to our design: `Order` and `OrderLine`.

First, is Order an Entity or a Value Object? Obviously it's an Entity—the details of the order may change over time, but it's the same order.

What about an OrderLine, though? If we change the quantity of a particular order line, for example, is it still the same order line? In most designs, it would make sense to say yes, it *is* still the same order line, even though the quantity or price has changed over time. So OrderLine is an Entity too, with its own identifier.

But now here's a question: if you change an order line, have you also changed the order that it belongs to?

In this case, it's clear that the answer is yes: changing a line also changes the entire order. In fact, having immutable data structures makes this unavoidable. If I have an immutable Order containing immutable OrderLines, then just making a copy of one of the order lines *does not* also make a copy of the Order as well. In order to make a change to an OrderLine contained in an Order, I need to make the change at the level of the Order, not at the level of the OrderLine.

For example, here's some pseudocode for updating the price of an order line:

```
/// We pass in three parameters:
/// * the top-level order
/// * the id of the order line we want to change
/// * the new price
let changeOrderLinePrice order orderLineId newPrice =

  // 1. find the line to change using the orderLineId
  let orderLine = order.OrderLines |> findOrderLine orderLineId

  // 2. make a new version of the OrderLine with the new price
  let newOrderLine = {orderLine with Price = newPrice}

  // 3. create a new list of lines, replacing
  //    the old line with the new line
  let newOrderLines =
    order.OrderLines |> replaceOrderLine orderLineId newOrderLine

  // 4. make a new version of the entire order, replacing
  //    all the old lines with the new lines
  let newOrder = {order with OrderLines = newOrderLines}

  // 5. return the new order
  newOrder
```

The final result, the output of the function, is a new Order containing a new list of lines, where one of the lines has a new price. You can see that immutability causes a ripple effect in a data structure, whereby changing one low-level component can force changes to higher-level components too.

Therefore, even though we're just changing one of its "subentities" (an OrderLine), we always have to work at the level of the Order itself.

This is a very common situation: we have a collection of Entities, each with their own ID and also some "top-level" Entity that contains them. In DDD terminology, a collection of Entities like this is called an *aggregate*, and the top-level Entity is called the *aggregate root*. In this case, the aggregate comprises both the Order and the collection of OrderLines, and the aggregate root is the Order itself.

Aggregates Enforce Consistency and Invariants

An aggregate plays an important role when data is updated. The aggregate acts as the consistency boundary: when one part of the aggregate is updated, other parts might also need to be updated to ensure consistency.

For example, we might extend this design to have an additional "total price" stored in the top-level Order. Obviously, if one of the lines changes price, the total must also be updated in order to keep the data consistent. This would be done in the changeOrderLinePrice function above. It's clear that the only component that "knows" how to preserve consistency is the top-level Order—the aggregate root—so this is another reason for doing all updates at the order level rather than at the line level.

The aggregate is also where any invariants are enforced. Say that you have a rule that every order has at least one order line. Then if you try to delete multiple order lines, the aggregate ensures there is an error when there's only one line left.

We'll discuss this further in Chapter 6, *Integrity and Consistency in the Domain*, on page 103.

Aggregate References

Let's say we need information about the customer to be associated with an Order. The temptation might be to add the Customer as a field of an Order, like this:

```
type Order = {
  OrderId : OrderId
  Customer : Customer  // info about associated customer
  OrderLines : OrderLine list
  // etc
  }
```

But think about the ripple effect of immutability. If I change any part of the customer, I must also change the order as well. Is that really what we want?

A much better design is just to store a *reference* to the customer, not the whole customer record itself. That is, we would just store the CustomerId in the Order type, like this:

```
type Order = {
  OrderId : OrderId
  CustomerId : CustomerId  // reference to associated customer
  OrderLines : OrderLine list
  // etc
  }
```

In this approach, when we need the full information about the customer, we would get the CustomerId from the Order and then load the relevant customer data from the database separately, rather than loading it as part of the order.

In other words, the Customer and the Order are *distinct* and *independent* aggregates. They each are responsible for their own internal consistency, and the only connection between them is via the identifiers of their root objects.

This leads to another important aspect of aggregates: they are the basic unit of persistence. If you want to load or save objects from a database, you should load or save whole aggregates. Each database transaction should work with a *single* aggregate and not include multiple aggregates or cross aggregate boundaries. See *Transactions*, on page 262, for more information.

Similarly, if you want to serialize an object to send it down the wire, you always send whole aggregates, not parts of them.

Just to be clear, an aggregate is not just any collection of Entities. For example, a list of Customers is a collection of Entities, but it's not a DDD "aggregate," because it doesn't have a top-level Entity as a root and it isn't trying to be a consistency boundary.

Here's a summary of the important role of aggregates in the domain model:

- An aggregate is a collection of domain objects that can be treated as a single unit, with the top-level Entity acting as the "root."
- All of the changes to objects inside an aggregate must be applied via the top level to the root, and the aggregate acts as a consistency boundary to ensure that all of the data inside the aggregate is updated correctly at the same time.
- An aggregate is the atomic unit of persistence, database transactions, and data transfer.

As you can see, defining the aggregates is an important part of the design process. Sometimes Entities that are used together are part of the same aggregate (OrderLine and Order) and sometimes they're not (Customer and Order). This is where collaborating with domain experts is critical: only they can help you understand the relationships between Entities and the consistency boundaries.

We'll be seeing lots of aggregates in the course of this modeling process, so we'll be using this terminology from now on.

More Domain-Driven Design Vocabulary

Here are the new DDD terms that we've introduced in this chapter:

- A *Value Object* is a domain object without identity. Two Value Objects containing the same data are considered identical. Value Objects must be immutable: if any part changes, it becomes a different Value Object. Examples of Value Objects are names, addresses, locations, money, and dates.
- An *Entity* is a domain object that has an intrinsic identity that persists even as its properties change. Entity objects generally have an ID or key field, and two Entities with the same ID/key are considered to be the same object. Entities typically represent domain objects that have a life-span and a history of changes, such as a document. Examples of Entities are customers, orders, products, and invoices.
- An *aggregate* is a collection of related objects that are treated as a single component both to ensure consistency in the domain and to be used as an atomic unit in data transactions. Other Entities should only reference the aggregate by its identifier, which is the ID of the "top-level" member of the aggregate, known as the "root."

Putting It All Together

We've created a lot of types in the chapter, so let's step back and look at how they fit together as a whole, as a complete domain model.

First, we put all these types in a namespace called OrderTaking.Domain, which is used to keep these types separate from other namespaces. In other words, we're using a namespace in F# to indicate a DDD bounded context, at least for now.

```
namespace OrderTaking.Domain

// types follow
```

Then let's add the simple types.

```
// Product code related
type WidgetCode = WidgetCode of string
  // constraint: starting with "W" then 4 digits
type GizmoCode = GizmoCode of string
  // constraint: starting with "G" then 3 digits
type ProductCode =
  | Widget of WidgetCode
  | Gizmo of GizmoCode

// Order Quantity related
type UnitQuantity = UnitQuantity of int
type KilogramQuantity = KilogramQuantity of decimal
type OrderQuantity =
  | Unit of UnitQuantity
  | Kilos of KilogramQuantity
```

These are all Value Objects and don't need an identifier.

The order, on the other hand, has an identity that's maintained as it changes—it's an Entity—so we must model it with an ID. We don't know whether the ID is a string or an int or a Guid, but we know we need it, so let's use Undefined for now. We'll treat other identifiers the same way.

```
type OrderId = Undefined
type OrderLineId = Undefined
type CustomerId = Undefined
```

The order and its components can be sketched out now:

```
type CustomerInfo = Undefined
type ShippingAddress = Undefined
type BillingAddress = Undefined
type Price = Undefined
type BillingAmount = Undefined

type Order = {
  Id : OrderId             // id for entity
  CustomerId : CustomerId  // customer reference
  ShippingAddress : ShippingAddress
  BillingAddress : BillingAddress
  OrderLines : OrderLine list
  AmountToBill : BillingAmount
  }

and OrderLine = {
  Id : OrderLineId  // id for entity
  OrderId : OrderId
  ProductCode : ProductCode
  OrderQuantity : OrderQuantity
  Price : Price
  }
```

In the snippet above, we're using the and keyword to allow forward references to undeclared types. See the explanation in *Organizing Types in Files and Projects*, on page 73.

Let's now conclude with the workflow itself. The input for the workflow, the UnvalidatedOrder, will be built from the order form "as is," so it will contain only primitive types such as int and string.

```
type UnvalidatedOrder = {
  OrderId : string
  CustomerInfo : ...
  ShippingAddress : ...
  ...
  }
```

We need two types for the output of the workflow. The first is the events type for when the workflow is successful:

```
type PlaceOrderEvents = {
  AcknowledgmentSent : ...
  OrderPlaced : ...
  BillableOrderPlaced : ...
  }
```

The second is the error type for when the workflow fails:

```
type PlaceOrderError =
  | ValidationError of ValidationError list
  | ...  // other errors

and ValidationError = {
    FieldName : string
    ErrorDescription : string
    }
```

Finally, we can define the top-level function that represents the order-placing workflow:

```
/// The "Place Order" process
type PlaceOrder =
  UnvalidatedOrder -> Result<PlaceOrderEvents,PlaceOrderError>
```

Obviously, lots of details still need to be fleshed out, but the process for doing that should now be clear.

Our model of the order-taking workflow isn't complete, though. For example, how are we going to model the different states of the order: validated, priced, and so on?

The Challenge Revisited: Can Types Replace Documentation?

At the beginning of this chapter, we gave ourselves a challenge: could we capture the domain requirements in the type system and in such a way that it can be reviewed by domain experts and other non-developers?

Well, if we look at the domain model listed above, we should be pleased. We have a complete domain model, documented as F# types rather than as text, but the types that we have designed look almost identical to the domain documentation that we developed earlier using *AND* and *OR* notation.

Imagine that you are a non-developer. What would you have to learn in order to understand this code as documentation? You'd have to understand the syntax for simple types (single-case unions), *AND* types (records with curly braces), *OR* types (choices with vertical bars), and "processes" (input, output, and arrows), but not much more. It certainly is more readable than a conventional programming language such as C# or Java.

Wrapping Up

In this chapter, we learned how to use the F# type system to model the domain using simple types, record types, and choice types. Throughout, we used the ubiquitous language of the domain, such as `ProductCode` and `OrderQuantity`, rather than developer-centric words such as `string` and `int`. Not once did we define a `Manager` or `Handler` type!

We also learned about different kinds of identity and how to model the DDD concepts of Value Object and Entity using types. And we were introduced to the concept of an "aggregate" as a way to ensure consistency.

We then created a set of types that looked very similar to the textual documentation at the beginning of this chapter. The big difference is that all these type definitions are *compilable code* and can be included with the rest of the code for the application. This in turn means that the application code is *always* in sync with the domain definitions, and if any domain definition changes, the application will fail to compile. We don't need to try to keep the design in sync with the code—the design *is* the code!

This approach, using types as documentation, is very general, and it should be clear how you can apply it to other domains as well. Because there's no implementation at this point, it's a great way to try ideas out quickly when you are collaborating with domain experts. And of course, because it is just

text, domain experts can review it easily without needing special tools, and maybe even write some types themselves!

We haven't yet addressed a few aspects of the design, though. How do we ensure that simple types are always constrained correctly? How can we enforce the integrity of aggregates? How are we going to model the different states of the order? These topics will be addressed in the next chapter.