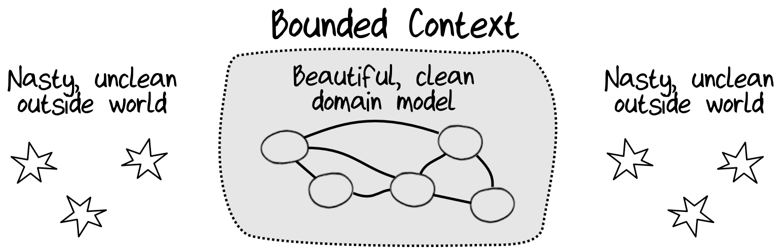
CHAPTER 6

Integrity and Consistency in the Domain

In the previous chapter, we looked at the basics of domain modeling using the F# type system. We built up a rich set of types that represented the domain but were also compilable and could be used to guide the implementation.

Since we've gone to this trouble to model the domain properly, we should take some precautions to make sure that any data in this domain is valid and consistent. The goal is to create a bounded context that always contains data we can trust as distinct from the untrusted outside world. If we can be sure that all data values are always valid, the implementation can stay clean and we can avoid having to do defensive coding.


Bounded Context
Nasty, unclean outside world
Beautiful, clean domain model
Nasty, unclean outside world

In this chapter, we'll look at modeling two aspects of a trusted domain: *integrity* and *consistency*.

Integrity (or validity) in this context means that a piece of data follows the correct business rules. For example:

- We said that a `UnitQuantity` is between 1 and 1000. Do we have to check this multiple times in our code, or can we rely on this to always be true?
- An order must always have at least one order line.
- An order must have a validated shipping address before being sent to the shipping department.

Consistency here means that different parts of the domain model agree about facts. Here are some examples:

- The total amount to bill for an order should be the sum of the individual lines. If the total differs, the data is inconsistent.
- When an order is placed, a corresponding invoice must be created. If the order exists but the invoice doesn't, the data is inconsistent.
- If a discount voucher code is used with an order, the voucher code must be marked as used so it can't be used again. If the order references that voucher but the voucher is not marked as used, the data is inconsistent.

How can we ensure this kind of data integrity and consistency? These are the kinds of questions we'll look at in this chapter. As always, the more information we can capture in the type system, the less documentation is needed and the more likely the code will be implemented correctly.

The Integrity of Simple Values

In the earlier discussion on modeling simple values on page 79, we saw that they should not be represented by string or int but by domain-focused types such as WidgetCode or UnitQuantity.

But we shouldn't stop there, because it's very rare to have an unbounded integer or string in a real-world domain. Almost always, these values are constrained in some way:

- An OrderQuantity might be represented by a signed integer, but it's very unlikely that the business wants it to be negative, or four billion.
- A CustomerName may be represented by a string, but that doesn't mean that it should contain tab characters or line feeds.

In our domain, we've seen some of these constrained types already. WidgetCode strings had to start with a specific letter, and UnitQuantity had to be between 1 and 1000. Here's how we've defined them so far, with a comment for the constraint.

```
type WidgetCode = WidgetCode of string   // starting with "W" then 4 digits
type UnitQuantity = UnitQuantity of int  // between 1 and 1000
type KilogramQuantity = KilogramQuantity of decimal // between 0.05 and 100.00
```

Rather than having the user of these types read the comments, we want to ensure that values of these types cannot be created unless they satisfy the constraints. Thereafter, because the data is immutable, the inner value never needs to be checked again. You can confidently use a WidgetCode or a UnitQuantity everywhere without ever needing to do any kind of defensive coding.

Sounds great. So how do we ensure that the constraints are enforced?

Answer: The same way we would in any programming language—make the constructor private and have a separate function that creates valid values and rejects invalid values, returning an error instead. In FP communities, this is sometimes called the *smart constructor* approach. Here's an example of this approach applied to UnitQuantity:

```
type UnitQuantity = private UnitQuantity of int
//                  ^ private constructor
```

So now a UnitQuantity value can't be created from outside the containing module due to the private constructor. However, if we write code in the same module that contains the type definition above, then we *can* access the constructor.

Let's use this fact to define some functions that will help us manipulate the type. We'll start by creating a submodule with exactly the same name (UnitQuantity); and within that, we'll define a create function that accepts an int and returns a Result type (as discussed in *Modeling Errors*) to return a success or a failure. These two possibilities are made explicit in its function signature: int -> Result<UnitQuantity,string>.

```
// define a module with the same name as the type
module UnitQuantity =

  /// Define a "smart constructor" for UnitQuantity
  /// int -> Result<UnitQuantity,string>
  let create qty =
    if qty < 1 then
      // failure
      Error "UnitQuantity can not be negative"
    else if qty > 1000 then
      // failure
      Error "UnitQuantity can not be more than 1000"
    else
      // success -- construct the return value
      Ok (UnitQuantity qty)
```

Compatibility with Older Versions of F#

Modules with the same name as a non-generic type will cause an error in versions of F# before v4.1 (VS2017), so you'll need to change the module definition to include a CompilationRepresentation attribute like this:

```
type UnitQuantity = ...

[<CompilationRepresentation(CompilationRepresentationFlags.ModuleSuffix)>]
module UnitQuantity =
  ...
```

One downside of a private constructor is that you can no longer use it to pattern-match and extract the wrapped data. One workaround for this is to define a separate value function, also in the UnitQuantity module, that extracts the inner value.

```
/// Return the wrapped value
let value (UnitQuantity qty) = qty
```

Let's see how this all works in practice. First, if we try to create a UnitQuantity directly, we get a compiler error:

```
let unitQty = UnitQuantity 1
//            ^ The union cases of the type 'UnitQuantity'
//              are not accessible
```

But if we use the UnitQuantity.create function instead, it works and we get back a Result, which we can then match against:

```
let unitQtyResult = UnitQuantity.create 1

match unitQtyResult with
| Error msg ->
  printfn "Failure, Message is %s" msg
| Ok uQty ->
  printfn "Success. Value is %A" uQty
  let innerValue = UnitQuantity.value uQty
  printfn "innerValue is %i" innerValue
```

If you have many constrained types like this, you can reduce repetition by using a helper module that contains the common code for the constructors. We don't have space to show that here, but there's an example in the Domain.SimpleTypes.fs file in the sample code for this book.

Finally, it's worth saying that using private is not the only way to hide constructors in F#. There are other techniques, such as using signature files, but we won't discuss them here.

Units of Measure

For numeric values, another way of documenting the requirements while ensuring type-safety is to use *units of measure*. With a units of measure approach, numeric values are annotated with a custom "measure." For example, we might define some units of measure for kg (kilogram) and m (meter) like this:

```
[<Measure>]
type kg

[<Measure>]
type m
```

And then we annotate some values with those units of measure like this:

```
let fiveKilos = 5.0<kg>
let fiveMeters = 5.0<m>
```

You don't need to define measure types for all the SI units. They are available in the Microsoft.FSharp.Data.UnitSystems.SI namespace.

Once this is done, the compiler will enforce compatibility between units of measure and present an error if they don't match.

```
// compiler error
fiveKilos = fiveMeters
//          ^ Expecting a float<kg> but given a float<m>

let listOfWeights = [
  fiveKilos
  fiveMeters  // <-- compiler error
  //             The unit of measure 'kg'
  //             does not match the unit of measure 'm'
  ]
```

In our domain, we could use units of measure to enforce that KilogramQuantity really *was* kilos, so that you couldn't accidentally initialize it with a value in pounds. We could encode this in the type like this:

```
type KilogramQuantity = KilogramQuantity of decimal<kg>
```

We've now got *two* checks: <kg> ensures that the number has the right unit, and KilogramQuantity enforces the constraints on the maximum and minimum values. This is probably design overkill for our particular domain, but it might be useful in other situations.

Units of measure need not just be used for physical units. You could use them to document the correct unit for timeouts (to avoid mixing up seconds and milliseconds) or for spatial dimensions (to avoid mixing up *x*- and *y*-axes), or for currency, and so on.

There's no performance hit from using units of measure. They're only used by the F# compiler and have no overhead at runtime.

Enforcing Invariants with the Type System

An *invariant* is a condition that stays true no matter what else happens. For example, at the beginning of the chapter, we said that a UnitQuantity must always be between 1 and 1000. That's an example of an invariant.

We also said that there must always be at least one order line in an order. Unlike the UnitQuantity case, this is an example of an invariant that can be captured directly in the type system. To make sure that a list isn't empty, we just need to define a NonEmptyList type. It's not built into F#, but it's easy to define yourself:

```
type NonEmptyList<'a> = {
  First: 'a
  Rest: 'a list
  }
```

The definition itself requires that there must always be at least one element, so a NonEmptyList is guaranteed never to be empty.

Of course, you'll also need some helper functions, such as add, remove, and so on. You can define these yourself or use one of the third-party libraries that provide this type, such as FSharpx.Collections.[1]

The Order type can now be rewritten to use this type instead of the normal list type:

```
type Order = {
  ...
  OrderLines : NonEmptyList<OrderLine>
  ...
  }
```

With this change, the constraint that "there is always at least one order line in an order" is now enforced automatically. Self-documenting code, and we've just eliminated any need to write unit tests for the requirement.

Capturing Business Rules in the Type System

Let's look at another modeling challenge: can we document business rules using just the type system? That is, we'd like to use the F# type system to represent what is valid or invalid so that the compiler can check it for us, instead of relying on runtime checks or code comments to ensure the rules are maintained.

Here's a real-world example. Suppose our company, Widgets Inc, stores email addresses for its customers. But let's also suppose not all email addresses should be treated the same way. Some email addresses have been *verified*—that is, the customer got a verification email and clicked on the verification link—while other email addresses aren't verified and we can't be sure they're valid. Furthermore, say some business rules are based on this difference, such as these:

1. https://fsprojects.github.io/FSharpx.Collections/

- You should only send verification emails to *unverified* email addresses (to avoid spamming existing customers).
- You should only send password-reset emails to *verified* email addresses (to prevent a security breach).

Now, how can we represent the two different situations in the design? A standard approach is to use a flag to indicate whether verification happened, like this:

```
type CustomerEmail = {
  EmailAddress : EmailAddress
  IsVerified : bool
  }
```

But this approach has a number of serious problems. First, it's not clear when or why the IsVerified flag should be set or unset. For example, if the customer's email address changes, it should be set back to false (because the new email is not yet verified). However, nothing in the design makes that rule explicit. It would be easy for a developer to accidentally forget to do this when the email is changed, or worse, be unaware of the rule altogether (because it's buried in some comments somewhere).

There's also the possibility of a security breach. A developer could write code that accidentally set the flag to true, even for an unverified email, which would allow password reset emails to be sent to unverified addresses.

So, what's a better way of modeling this?

The answer is, as always, to pay attention to the domain. When domain experts talk about "verified" and "unverified" emails, you should model them as separate things. In this case, when a domain expert says "a customer's email is either verified or unverified," we should model that as a choice between two types, like this:

```
type CustomerEmail =
  | Unverified of EmailAddress
  | Verified of EmailAddress
```

But that still doesn't prevent us from accidentally creating the Verified case by passing in an unverified EmailAddress. To solve that problem, we'll do what we always do and create a new type! In particular, we'll create a type VerifiedEmailAddress, which is different from the normal EmailAddress type. Now our choice looks like this:

```
type CustomerEmail =
  | Unverified of EmailAddress
  | Verified of VerifiedEmailAddress // different from normal EmailAddress
```

Here's the clever part: we can give VerifiedEmailAddress a *private* constructor so normal code can't create a value of that type—only the verification service can create it.

That means that if I have a new email address, I *have* to construct a CustomerEmail using the Unverified case because I don't have a VerifiedEmailAddress. The only way I can construct the Verified case is if I have a VerifiedEmailAddress, and the only way I can get a VerifiedEmailAddress is from the email verification service itself.

This is an example of the important design guideline, "Make illegal states unrepresentable." We're trying to capture business rules in the type system. If we do this properly, invalid situations can't ever exist in the code and we never need to write unit tests for them. Instead, we have "compile-time" unit tests.

Another important benefit of this approach is that it actually documents the domain better. Rather than having a simplistic EmailAddress that tries to serve two roles, we have two distinct types with different rules around them. And typically, once we have created these more fine-grained types, we immediately find uses for them.

For example, I can now explicitly document that the workflow that sends a password-reset message *must* take a VerifiedEmailAddress parameter as input rather than a normal email address.

```
type SendPasswordResetEmail = VerifiedEmailAddress -> ...
```

With this definition, we don't have to worry about someone accidentally passing in a normal EmailAddress and breaking the business rule because they haven't read the documentation.

Here's another example. Let's say we have a business rule that we need some way of contacting a customer:

- "A customer must have an email or a postal address."

How should we represent this? The obvious approach is just to create a record with both an Email and an Address property, like this:

```
type Contact = {
  Name: Name
  Email: EmailContactInfo
  Address: PostalContactInfo
  }
```

But this is an incorrect design. It implies both Email and Address are required. So let's make them optional:

```
type Contact = {
  Name: Name
  Email: EmailContactInfo option
  Address: PostalContactInfo option
  }
```

But this isn't correct either. As it stands, Email and Address could both be missing, and that would break the business rule.

Now, of course, we could add special runtime validation checks to make sure that this couldn't happen. But can we do better and represent this in the type system? Yes, we can!

The trick is to look at the rule closely. It implies that a customer has the following:

- An email address only
- A postal address only
- Both an email address and a postal address

That's only three possibilities. How can we represent these three? With a choice type, of course!

```
type BothContactMethods = {
  Email: EmailContactInfo
  Address : PostalContactInfo
  }

type ContactInfo =
    | EmailOnly of EmailContactInfo
    | AddrOnly of PostalContactInfo
    | EmailAndAddr of BothContactMethods
```

And then we can use this choice type in the main Contact type, like this:

```
type Contact = {
  Name: Name
  ContactInfo : ContactInfo
  }
```

Again what we've done is good for developers (we can't accidentally have no contact information—one less test to write), but it's also good for the design. The design makes it very clear that only three cases are possible and exactly what those three cases are. We don't need to look at documentation; we can just look at the code itself.

Making Illegal States Unrepresentable in Our Domain

Are there any places in our design where we can put this approach into practice?

I can think of one aspect of the design that is very similar to the email validation example. In the validation process, we documented that there were unvalidated postal addresses (such as UnvalidatedAddress) and validated postal addresses (ValidatedAddress).

We could ensure that we never mix up these two cases and also ensure that we use the validation function properly by doing the following:

- Create two distinct types: UnvalidatedAddress and ValidatedAddress
- Give the ValidatedAddress a private constructor and then ensure that it can only be created by the address validation service.

```
type UnvalidatedAddress = ...

type ValidatedAddress = private ...
```

The validation service takes an UnvalidatedAddress and returns an optional ValidatedAddress (optional to show that validation might fail).

```
type AddressValidationService =
  UnvalidatedAddress -> ValidatedAddress option
```

To enforce the rule that an order must have a validated shipping address before being sent to the shipping department, we'll create two more distinct types (UnvalidatedOrder and ValidatedOrder) and require that a ValidatedOrder record contain a shipping address that is a ValidatedAddress.

```
type UnvalidatedOrder = {
  ...
  ShippingAddress : UnvalidatedAddress
  ...
  }

type ValidatedOrder = {
  ...
  ShippingAddress : ValidatedAddress
  ...
  }
```

And now we can guarantee, without ever writing a test, that addresses in a ValidatedOrder have been processed by the address validation service.

Consistency

So far in this chapter we've looked at ways to enforce the integrity of the data in the domain, so now let's finish up by taking a look at the related concept of *consistency*.

We saw some examples of consistency requirements at the beginning of the chapter:

- The total amount for an order should be the sum of the individual lines. If the total differs, the data is inconsistent.
- When an order is placed, a corresponding invoice must be created. If the order exists but the invoice doesn't, the data is inconsistent.
- If a discount voucher code is used with an order, the voucher code must be marked as used so it can't be used again. If the order references that voucher but the voucher is not marked as used, the data is inconsistent.

As described here, consistency is a business term, not a technical one, and what consistency means is always context-dependent. For example, if a product price changes, should any unshipped orders be immediately updated to use the new price? What if the default address of a customer changes? Should any unshipped orders be immediately updated with the new address? There's no right answer to these questions—it depends on what the business needs.

Consistency places a large burden on the design, though, and can be costly, so we want to avoid the need for it if we can. Often during requirements gathering, a product owner will ask for a level of consistency that is undesirable and impractical. In many cases, however, the need for consistency can be avoided or delayed.

Finally, it's important to recognize that consistency and atomicity of persistence are linked. There's no point, for example, in ensuring that an order is internally consistent if the order is not going to be persisted atomically. If different parts of the order are persisted separately and then one part fails to be saved, then anyone loading the order later will be loading an order that is not internally consistent.

Consistency Within a Single Aggregate

In *Domain Modeling with Types*, we introduced the concept of an aggregate and noted that it acts both as a consistency boundary and as a unit of persistence. Let's see how this works in practice.

Let's say that we require that the total amount for an order should be the sum of the individual lines. The easiest way to ensure consistency is simply to calculate information from the raw data rather than storing it. In this case then, we could just sum the order lines every time we need the total, either in memory or using a SQL query.

If we do need to persist the extra data (say an additional `AmountToBill` stored in the top-level `Order`), then we need to ensure that it stays in sync. In this case then, if one of the lines is updated, the total must also be updated in order to keep the data consistent. It's clear that the only component that "knows" how to preserve consistency is the top-level `Order`. This is a good reason for doing all updates at the order level rather that at the line level—the order is the aggregate that enforces a consistency boundary. Here's some code that demonstrates how this might work:

```
/// 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 =

  // find orderLine in order.OrderLines using orderLineId
  let orderLine = order.OrderLines |> findOrderLine orderLineId

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

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

  // make a new AmountToBill
  let newAmountToBill = newOrderLines |> List.sumBy (fun line -> line.Price)

  // make a new version of the order with the new lines
  let newOrder = {
      order with
        OrderLines = newOrderLines
        AmountToBill = newAmountToBill
      }

  // return the new order
  newOrder
```

Aggregates are also the unit of atomicity, so if we save this order to a relational database, say, we must ensure that the order header and the order lines are all inserted or updated in the same transaction.

Consistency Between Different Contexts

What if we need to coordinate between different contexts? Let's look at the second example on the list above:

- When an order is placed, a corresponding invoice must be created. If the order exists but the invoice doesn't, the data is inconsistent.

Invoicing is part of the billing domain, not the order-taking domain. Does that mean we need to reach into the other domain and manipulate its objects? No, of course not. We must keep each bounded context isolated and decoupled.

What about using the billing context's public API, like this:

```
Ask billing context to create invoice
If successfully created:
   create order in order-taking context
```

This approach is much trickier than it might seem, because you need to handle either update failing. There are ways to synchronize updates across separate systems properly (such as a two-phase commit), but in practice it's rare to need this. In his article "Starbucks Does Not Use Two-Phase Commit,"[2] Gregor Hohpe points out that in the real world businesses generally do not require that every process move in lockstep, waiting for all subsystems to finish one stage before moving to the next stage. Instead, coordination is done asynchronously using messages. Occasionally, things will go wrong, but the cost of dealing with rare errors is often much less than the cost of keeping everything in sync.

For example, let's say that instead of requiring an invoice be created immediately, we just send a message (or an event) to the billing domain and then continue with the rest of the order processing.

So now what happens if that message gets lost and no invoice is created?

- One option is to do nothing. Then the customer gets free stuff and the business has to write off the cost. That might be a perfectly adequate solution if errors are rare and the costs are small (as in a coffee shop).
- Another option is to detect that the message was lost and resend it. This is basically what a reconciliation process does: compare the two sets of data, and if they don't match up, fix the error.
- A third option is to create a *compensating action* that "undoes" the previous action or fixes the error. In an order-taking scenario, this would be equivalent to cancelling the order and asking the customer to send the items back! More realistically, a compensating action might be used to do things such as correct mistakes in an order or issue refunds.

In all three cases, there's no need for rigid coordination between the bounded contexts.

If we have a requirement for consistency, then we need to implement the second or third option. But this kind of consistency won't take effect immediately. Instead, the system will become consistent only after some time has passed—"eventual consistency." Eventual consistency is not "optional consistency": it's still very important that the system be consistent at some point in the future.

2. http://www.enterpriseintegrationpatterns.com/ramblings/18_starbucks.html

Here's an example. Let's say that if a product price has changed, we want to update the price for all orders that haven't shipped yet. If we need immediate consistency, then when we update the price in the product record, we also have to update all affected orders and do all this within the same transaction. This could take some time. But if we don't require instant consistency when a product price has changed, we might instead create a `PriceChanged` event that in turn triggers a series of `UpdateOrderWithChangedPrice` commands to update the outstanding orders. These commands will be processed some time after the price in the product record has changed, perhaps seconds later, perhaps hours later. Eventually the orders will be updated and the system will be consistent.

Consistency Between Aggregates in the Same Context

What about ensuring consistency between aggregates in the *same* bounded context? Let's say that two aggregates need to be consistent with each other. Should we update them together in the same transaction or update them separately using eventual consistency? Which approach should we take?

As always, the answer is that it depends. In general, a useful guideline is "only update one aggregate per transaction." If more than one aggregate is involved, we should use messages and eventual consistency as described above, even though both aggregates are within the same bounded context. But sometimes—and especially if the workflow is considered by the business to be a single transaction—it might be worth including all affected entities in the transaction. A classic example is transferring money between two accounts, where one account increases and the other decreases.

```
Start transaction
Add X amount to accountA
Remove X amount from accountB
Commit transaction
```

If the accounts are represented by an `Account` aggregate, then we would be updating two different aggregates in the same transaction. That's not necessarily a problem, but it might be a clue that you can refactor to get deeper insights into the domain. In cases like this, for example, the transaction often has its own identifier, which implies that it's a DDD Entity in its own right. In that case, why not model it as such?

```
type MoneyTransfer = {
  Id: MoneyTransferId
  ToAccount : AccountId
  FromAccount : AccountId
  Amount: Money
  }
```

After this change, the Account entities would still exist, but they would no longer be directly responsible for adding or removing money. Instead the current balance for an Account would now be calculated by iterating over the MoneyTransfer records that reference it. We've not only refactored the design, but we've also learned something about the domain.

This also shows that you shouldn't feel obligated to reuse aggregates if it doesn't make sense to do so. If you need to define a new aggregate just for one use case, go ahead.

Multiple Aggregates Acting on the Same Data

We stressed earlier that aggregates act to enforce integrity constraints, so how can we ensure that the constraints are enforced consistently if we have multiple aggregates that act on the same data? For example, we might have an Account aggregate and a MoneyTransfer aggregate that are both acting on account balances and both needing to ensure that a balance doesn't become negative.

In many cases constraints can be shared between multiple aggregates if they are modeled using types. For example, the requirement that an account balance never be below zero could be modeled with a NonNegativeMoney type. If this is not applicable, then you can use shared validation functions. This is one advantage of functional models over object-oriented models: validation functions are not attached to any particular object and don't rely on global state, so they can easily be reused in different workflows.

Wrapping Up

In this chapter, we learned how to ensure that data inside our domain could be trusted.

We saw that the combination of "smart constructors" for simple types, and "making illegal states unrepresentable" for more complex types, meant that we could enforce many kinds of integrity rules using the type system itself, leading to more self-documenting code and less need for unit tests.

We also looked at maintaining consistent data within one bounded context and between bounded contexts, concluding that, unless you are working within a single aggregate, you should design for eventual consistency rather that immediate consistency.

In the next chapter, we'll put all this into practice as we model our order-placing workflow.