**Towards an immutable domain model**

# Introduction

This is the first part of a (short) series of blogs on implementing a [rich domain model](http://domaindrivendesign.org/) using only *immutable domain objects*. This first part introduces a (simple) example domain model and provides a [JPA](http://en.wikipedia.org/wiki/Java_Persistence_API) implementation using Scala. This example will serve as a baseline that should be familiar to most developers who have experience with an [ORM](http://en.wikipedia.org/wiki/Object-relational_mapping). The other parts of this series will redesign the example to use only immutable objects and will explore some of the benefits and drawbacks of doing so.

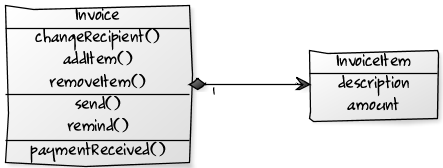
**Why immutability?**

So why even try to achieve immutability in your domain classes? What makes immutable objects useful compared to mutable ones?

* Immutable objects are often easier to use. Compare [java.util.Calendar](http://download.oracle.com/javase/6/docs/api/java/util/Calendar.html)(mutable) with Joda-Time’s [DateTime](http://joda-time.sourceforge.net/apidocs/org/joda/time/DateTime.html) (immutable).
* Immutable objects reduce the number of possible interactions (aliasing) between different parts of the program. Just consider how hard it would be to program Java if the String class was mutable and all methods provided by String mutated the receiving instance and returned void. Sharing these strings with other parts of the program could be very dangerous.
* Immutable objects can be safely shared between multiple threads. An example would be the implementation of a shared cache: immutable objects make this quite a bit easier.
* Implementing an immutable object is often easier, as there is less that can go wrong and the design space is “smaller”.

Furthermore the rising availability (and popularity?) of functional programming languages (such as Clojure, F#, Haskell, and Scala) make it easier to work with immutable data than imperative languages such as Java. This can have important implications on how we design business applications, so it is worth exploring making the domain model immutable, especially since a business domain model is often considered to be *inherently* mutable.

**A not “totally trivial” example**

[](http://blog.zilverline.com/wp-content/uploads/2011/01/526f3986.png)

The example is a (simplified) version of anInvoice. Invoices consist of a recipient, invoice items, and the invoice amount. The lifecycle of our simplified invoice consists of the following stages:

1. Draft invoices start out empty and can be freely edited
2. Once all required information is present the invoice can be *send*
3. A sent invoice can receive payment. When payment is not received before the due date we can send a reminder.

The diagram to the right depicts the main interface of our Invoice class. The methods are grouped in relation to the state of the invoice.

The full JPA implementation is available as [Invoice.scala](http://github.com/erikrozendaal/immutable-domain-example/blob/master/src/main/scala/com/zilverline/examples/immutabledomain/jpa/Invoice.scala) on github.[Test cases](http://github.com/erikrozendaal/immutable-domain-example/blob/master/src/test/scala/com/zilverline/examples/immutabledomain/jpa/JpaInvoiceSpec.scala) are also available. Below you’ll find a small excerpt from the Invoice implementation:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | @OneToMany(cascade = Array(CascadeType.ALL))  @OrderBy  private var \_items: List[InvoiceItem] = new ArrayList    @Basic(optional = false)  private var \_totalAmount: BigDecimal = BigDecimal.ZERO    @Temporal(TemporalType.DATE)  private var \_sentDate: Date = \_    def sent\_? = \_sentDate != null    def removeItem(index: Int) {    require(!sent\_?,  "items cannot be changed after invoice is sent")    val item = \_items.remove(index)    \_totalAmount = \_totalAmount.subtract(item.amount)  } |

Even for such a simplified example the resulting code is already quite large and feels messy. The mix of low-level data manipulation (JPA annotations, conversions between Joda-Time dates and the Java dates supported by JPA, use of null instead of Options) with the high-level behavioural code seems to be a clear violation of the [Single Responsibility Principle](http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod) (SRP).

This is because we are trying to meet three separate needs that are common to many business applications inside a single class:

1. provide a record of all “important” data ([*durability*](http://en.wikipedia.org/wiki/Durability_(database_systems)))
2. implement business rules and logic (*behaviour*)
3. answer questions about aninvoice (*reporting*)

The JPA implementation of the example makes various compromises with regards to these three needs:

1. The “total amount” field is denormalized to make it easier to perform queries
2. Only the last reminder date is kept and previous reminder dates are overwritten

Besides being messy, there is also something strange about hiding the data of anInvoice behind a behavioural interface, but then fully exposing this data to the ORM, as well as any queries and reports that access this information. Furthermore “getters” are provided so the data can be used by clients. Encapsulation should make it possible to easily change the implementation without any of the user of anInvoice being aware of this, but anyone who has tried to actually do this with JPA backed objects knows it is not as easy as it should be!

Is it possible to split this object into three different ones, each with just a single responsibility? Certainly, and many existing systems already do so. The Invoice entity can be implemented as an [anemic domain](http://en.wikipedia.org/wiki/Anemic_Domain_Model) object, and is used to provide the required durability. The behavioral aspects are implemented by controller, manager, or service objects, and the reporting needs are provided by a [DAO](http://en.wikipedia.org/wiki/Data_access_object). This often results in procedural *transaction scripts* that implement the behavioural requirements by mutating the “durable” entity. Although we could redesign our Invoice this way, it only seems to lead us further from an immutable domain model.

In the next part we’ll look at a different way meet these three basic needs which do allow an immutable implementation of the domain model. The name of the game will be the addictive combination of [CQRS](http://cqrsinfo.com/) and [Event Sourcing](http://martinfowler.com/eaaDev/EventSourcing.html).

1. **Immutable change**

In the first [part](http://blog.zilverline.com/2011/02/01/towards-an-immutable-domain-model-introduction-part-1/) of this series an Invoice domain object was defined as a starting point for discussing immutable domain objects. JPA and Scala were used for the example implementation. In this part we’ll look at this example from a different perspective to move closer to an immutable domain model.

**“Inherent mutability”**

One of the first misconceptions that need addressing is that an invoice is somehow “inherently mutable”. This may true in the “real world” (where an invoice might be a piece of paper that anyone just writes on), but when it comes to the models we use to build software systems, there is really no need to let the real world limit our designs.

The first step is to make the implicit notion of “change-over-time” (mutability) explicit in our model. The trick is to model *all* state changes of the invoice explicitly as an *event*. This event becomes our explicit notion of change. Using events to model all state changes is also known as [event sourcing](http://martinfowler.com/eaaDev/EventSourcing.html), journaling or transaction logging. Notice that this is nothing new: accountancy has been doing this for many centuries!

For our invoice example we can define the following events:

case class InvoiceItem(id: Int, description: String, amount: BigDecimal)

sealed trait InvoiceEvent {

val invoiceId: Int

}

case class InvoiceCreated(invoiceId: Int) extends InvoiceEvent

case class InvoiceRecipientChanged(invoiceId: Int, recipient: Option[String]) extends InvoiceEvent

case class InvoiceItemAdded(invoiceId: Int, item: InvoiceItem, totalAmount: BigDecimal) extends InvoiceEvent

case class InvoiceItemRemoved(invoiceId: Int, item: InvoiceItem, totalAmount: BigDecimal) extends InvoiceEvent

case class InvoiceSent(invoiceId: Int, sentDate: LocalDate, dueDate: LocalDate) extends InvoiceEvent

case class InvoiceReminderSent(invoiceId: Int, reminderDate: LocalDate) extends InvoiceEvent

case class InvoicePaymentReceived(invoiceId: Int, paymentDate: LocalDate) extends InvoiceEvent

We can then use these events to describe any valid invoice, for example:

1. Invoice 17 created
2. Invoice 17’s recipient changed to “Erik”
3. Item added to invoice 17 with description “Beverage” and amount 2.95
4. Invoice 17 sent on 2011-2-1 with payment due date 2011-2-15
5. Payment received for invoice 17 on 2011-2-13

Notice that these events are all named using the past tense. The events represent the *results* of behaviours that have already happened, not the behaviours themselves. By defining these events we have simultaneously reduced the number of possible mutations and made it easier for business domain experts to understand the system. We’ve raised the level of abstraction.

Also notice that the use of events to capture important facts about the business domain is a perfect match for the *durability* need. Since events are immutable and never deleted, you can be much more certain that no important information is ever lost, unlike our JPA example which overwrites the reminder date whenever a new reminder is sent, or where bugs could easily lead to corruption of supposedly durable data.

See [this presentation](http://skillsmatter.com/podcast/agile-testing/greg-young-cqrs-event-sourcing-the-business-perspective) by Greg Young for much more business related benefits.

**Event sourced invoice implementation**

So let’s move on to the implementation of an invoice using event sourcing. To do this we need to make two major changes to our implementation:

1. Change the behaviour to *generate* and *track* events instead of mutating state directly
2. Restore the state of the Invoice from its historical events.

Let’s capture this in the following trait:

trait AggregateRoot[Event] {

protected def applyEvent: Event => Unit

def uncommittedEvents: Iterable[Event] = \_uncommittedEvents

def markCommitted = \_uncommittedEvents.clear

def loadFromHistory(history: Iterable[Event]) = history.foreach(applyEvent)

protected def record(event: Event) {

applyEvent(event)

\_uncommittedEvents += event

}

private val \_uncommittedEvents = mutable.Queue[Event]()

}

The trait is parameterized (generic) over the type of events the aggregate root can handle. The first declared method (applyEvent) is *abstract*. This method is used to update the current state according to the given event and returns no value of interest (Unit, the Scala equivalent of void).

The next three methods (uncommittedEvents, markCommitted, andloadFromHistory) allow the clients of our aggregate root to load from and persist to a durable store. The uncommittedEvents are stored in a mutable collection while the loadFromHistory method simply applies (plays back) each event in the history to the current instance.

Finally the record method allows our Invoice implementation to update its current state *and* record this change in the collection of uncommittedEvents.

An example usage (using [Scala Specs](http://code.google.com/p/specs/) syntax) can be found below. Here we first load an invoice’s history, invoke some behaviour, and check that correct event was generated:

"ready to send invoice" should {

"generate invoice sent event" in {

val invoice = new Invoice

invoice.loadFromHistory(Seq(

InvoiceCreated(1),

InvoiceRecipientChanged(1, Some("Erik")),

InvoiceItemAdded(1, InvoiceItem(1, "Food", 2.95), 2.95)))

invoice.send

invoice.uncommittedEvents must contain(

InvoiceSent(1,

sentDate = new LocalDate(2011, 1, 29),

dueDate = new LocalDate(2011, 2, 12)))

}

}

The full implementation of the event sourced Invoice can be found at[Invoice.scala](http://github.com/erikrozendaal/immutable-domain-example/blob/master/src/main/scala/com/zilverline/examples/immutabledomain/eventsourcing/Invoice.scala). Let’s take a look at the send method:

def send {

require(!sent\_?, "invoice already sent")

require(readyToSend\_?, "recipient and items must be specified before sending")

val now = new LocalDate

record(InvoiceSent(id, sentDate = now, dueDate = now.plusDays(14)))

}

The method first checks the current state to see if the invoice is ready to be sent. If so, it calls the record method (defined in the AggregateRoot trait) with a new InvoiceSent event. The record method will store the event in the uncommittedEvents collection and invoke the Invoice’s applyEvent method, of which an excerpt is listed here:

protected def applyEvent = {

// [... code omitted ...]

case event: InvoiceSent =>

sent\_? = true

dueDate = Some(event.dueDate)

// [... code omitted ...]

}

One important thing to notice is that since the Invoice implementation is no longer concerned with durability or reporting needs, we only need to track the state necessary to fulfil the behavioural contract. For example, we simply use a Boolean flag to remember that the invoice was sent. We don’t need the actual sent date, since no behaviour currently requires it. The same goes for various other fields, such as the recipient’s name or the payment date. However, the due date *is* needed, as it is used to check if we can send a reminder.

So now that the domain model is freed of the durability and reporting responsibilities, it has become both a smaller and more focused implementation of the behavioural needs, compared to the original JPA implementation.

**Reports and queries**

I won’t be going into the details on reporting in this series, but it should be clear that it is easy to define any kind of report based on the events generated by the domain. These reports can even be created many years after the original events were generated, allowing you to define new ways to look at historical data.

**Conclusion**

By splitting the JPA Invoice class into three different parts, aligned by the needs for durability, reporting, and behaviour, each part becomes easier to implement and is better suited to the need.

There is still mutable state, but the mutations are now isolated into the applyEvent method and each change is now an explicit, immutable event. In the next parts we’ll look at making the invoice immutable and how that can help us in the design of the domain model.

# Immutability achieved

In [part 1](http://blog.zilverline.com/2011/02/01/towards-an-immutable-domain-model-introduction-part-1/) we looked at a stereotypical implementation of an Invoice domain class. In [part 2](http://blog.zilverline.com/2011/02/02/towards-an-immutable-domain-model-%E2%80%93-immutable-change-part-2/) we introduced event sourcing to extract the durability and reporting concerns from the behavioural requirements and to move closer to making the Invoice *immutable*. In this part we’ll explore the first implementation of an immutable Invoice.

**Mutable to immutable: a cheap trick**

For the first immutable implementation we’ll use a simple trick that can be used to turn any mutable structure into an immutable one: instead of modifying the existing instance to perform mutation, we’ll return a new copy of the existing instance with the modifications already applied to it. This leaves the original instance unchanged. Here’s a quick example of such a transformation applied to a mutable counter:

class MutableCounter(var current: Int) {

def increment {

current += 1

}

}

class ImmutableCounter(val current: Int) {

def increment = new ImmutableCounter(current + 1)

}

Notice that we changed the var into a val and the increment method was changed to return a new counter. So moving from a mutable to immutable implementation can basically be achieved by mechanical code translation. Maybe future IDEs will provide us with refactoring support for this?

**The problem of creation**

Before we take on the Invoice class we need to take care of some other details first. The previous Invoice implementation provides two ways of construction:

1. Construct a brand new invoice by passing a new invoice id to the constructor
2. Construct an invoice using the default constructor, which should only be used when reloading from durable storage

In the second case we temporarily have an “invalid” invoice. This is a bad idea, but unfortunately all too common. In fact, JPA *requires* a default constructor, making it harder to always enforce validity.

Although not strictly required, we’ll move creation into a factory. Both the factory and the invoice will use event sourcing, so we define EventSourced trait to extract commonality:

trait EventSourced[ES <: EventSourced[ES, Event], Event] {

def applyEvent: Event => ES

def unhandled(event: Event) = error("event " + event + " does not apply to " + this)

}

Notice that the applyEvent method has been changed from type Event => Unit to a method that takes an Event and returns a new instance of type ES. The unhandled method is just for convenience.

The definitions for the aggregate root and factory are listed below:

trait AggregateRoot[AR <: AggregateRoot[AR, Event], Event] extends EventSourced[AR, Event] {

def uncommittedEvents: List[Event]

def markCommitted: AR

}

trait AggregateFactory[AR <: AggregateRoot[AR, Event], Event] extends EventSourced[AR, Event] {

def loadFromHistory(history: Iterable[Event]): AR = {

var aggregate = applyEvent(history.head)

for (event <- history.tail)

aggregate = aggregate.applyEvent(event)

return aggregate.markCommitted

}

}

The AggregateRoot trait defines two abstract methods: uncommittedEvents should return the current list of uncommitted events, and markCommitted should return a new instance (remember, immutability!) of the aggregate root with the uncommitted events cleared.

The AggregateFactory provides the loadFromHistory method. It first applies the initial event to itself to create an instance of the aggregate and then applies the remaining events to the successive instances of the aggregate. The final instance is returned, but not before clearing the uncommitted events. We’ll see why later. (Exercise: reimplement loadFromHistory without the imperative for-loop. FoldLeft is your friend.)

**Immutable Invoice**

Now that we’ve taken care of the infrastructure, let’s start with our shiny new immutable Invoice. First the factory, which we’ll make a[companion object](http://daily-scala.blogspot.com/2009/09/companion-object.html). Here it is:

object Invoice extends AggregateFactory[Invoice, InvoiceEvent] {

def create(invoiceId: Int) = applyEvent(InvoiceCreated(invoiceId))

def applyEvent = {

case event: InvoiceCreated => Invoice(event :: Nil, event.invoiceId)

case event => unhandled(event)

}

}

The factory simply provides a way to create a new invoice from scratch using an InvoiceCreated event. The implemented applyEventmethod then instantiates a new Invoice for us with the provided id and the creation event as its only uncommitted event.

The immutable invoice class is similar, just bigger. Let’s first define the data it needs:

case class Invoice (

uncommittedEvents: List[InvoiceEvent],

id: Int,

recipient\_? : Boolean = false,

nextItemId: Int = 1,

items: Map[Int, InvoiceItem] = Map.empty,

sent\_? : Boolean = false,

paid\_? : Boolean = false,

dueDate: Option[LocalDate] = None)

extends AggregateRoot[Invoice, InvoiceEvent] {

// [... code omitted ...]

}

All data fields are now vals instead of vars and we’ve added the list of uncommitted events. The invoice has also been changed to a [case class](http://daily-scala.blogspot.com/2010/01/case-classes-in-28.html) so that we can use the convenient copy method in our applyEvent implementation.

Let’s take a look at the send method and the corresponding case in theapplyEvent method:

def send: Invoice = {

require(!sent\_?, "invoice already sent")

require(readyToSend\_?, "recipient and items must be specified before sending")

val now = new LocalDate

applyEvent(InvoiceSent(id, sentDate = now, dueDate = now.plusDays(14)))

}

def applyEvent = {

// [... code omitted ...]

case event: InvoiceSent =>

copy(event :: uncommittedEvents, sent\_? = true, dueDate = Some(event.dueDate))

// [... code omitted ...]

}

Compared to the previous implementation of [send](https://gist.github.com/806373#file_invoice_send1.scala) and [applyEvent](https://gist.github.com/806373#file_apply_event1.scala)very little has changed. The main differences are that we return a new copy of Invoice from send and that we explicitly prepend theInvoiceSent event to the list of uncommitted events.

Now that applyEvent always adds the event to the uncommitted events it also becomes clear why we need to invoke markCommitted when loading an invoice from its history. If we didn’t, all historical events would be part of the invoice’s uncommitted events after reloading!

Talking about markCommitted, we still need to implement it. It should return a new copy of the invoice with the list of uncommitted events emptied. That’s easy:

def markCommitted = copy(uncommittedEvents = Nil)

That’s it! The full implementation of the immutable invoice can be found at [Invoice.scala](https://github.com/erikrozendaal/immutable-domain-example/blob/master/src/main/scala/com/zilverline/examples/immutabledomain/immutable/Invoice.scala). Here’s an example on how you could use such an invoice in client code:

"ready to send invoice" should {

"generate invoice sent event" in {

val invoice = Invoice.create(1)

.changeRecipient(Some("Erik"))

.addItem("Food", 2.95)

.send

invoice.uncommittedEvents must contain(

InvoiceSent(1,

sentDate = new LocalDate(2011, 1, 29),

dueDate = new LocalDate(2011, 2, 12)))

}

}

That’s a pretty nice example of a [fluent interface](http://en.wikipedia.org/wiki/Fluent_interface), something quite common to functional code.

**Conclusion**

Compared to the switch from JPA to event sourcing, the move from mutability to immutability was quite straightforward. No major surgery was required. This is a good thing. It shows that it is possible, and easy, to implement your domain using immutable objects. We’ve also managed to make some minor improvements, such as moving loadFromHistory from the aggregate itself to a factory.

The main drawback is the need to explicitly deal with uncommitted events in applyEvent and the addition of the markCommitted method to each implementer of AggregateRoot.

In the next parts we’ll explore how the immutable implementation can help improve the design and implementation of the Invoice class and how we can remove some of the boilerplate code related to keeping track of uncommitted events.

# 4. Believe the type

We’re already on the fourth part of this series. In this part we’ll explore how we can improve on the original design now that the invoice is immutable.

**Cohesion**

Let’s take a look at the [cohesion](http://www.hokstad.com/why-coupling-is-always-bad-cohesion-vs-coupling.html) of current the Invoice implementation. We do this by looking at the public methods and the instance fields they affect. Here they are, grouped by pre-condition:

| **Pre-condition** | **Methods** | **Affects** |
| --- | --- | --- |
| Draft? | changeRecipient addItem removeItem | recipient\_? nextItemId items |
| Ready to sent? | send | sent\_? dueDate |
| Sent? | readyToPay\_? pay late\_? remind | paid\_? |
| Paid? | (none) | (none) |

So it looks like there are actually a couple of classes hiding inside our big Invoice class. Mostly related to the draft, sent, and paid state of the invoice.

Another improvement we can make is to ensure that the bad code listed below fails to compile, instead of blowing up at runtime:

|  |  |
| --- | --- |
| 1 | Invoice.create(2).changeRecipient(Some("Erik")).pay.send.addItem("Food", 2.95) |

**Believe the type**

Can we make the type system work for us and prohibit this code? Certainly! We have to split the current invoice class into a few subclasses where each subclass only defines the methods that make sense. We also need to change the public methods that return anInvoice to return an instance of the correct subtype instead.

Let’s see how this works out for the DraftInvoice class:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | sealed trait Invoice extends AggregateRoot[Invoice, InvoiceEvent]    case class DraftInvoice(      uncommittedEvents: List[InvoiceEvent],      id: Int,      recipient\_? : Boolean = false,      nextItemId: Int = 1,      items: Map[Int, InvoiceItem] = Map.empty)    extends Invoice {    // [... code omitted ...]  } |

Compared to the previous version of Invoice we managed to drop the three fields (send\_?, dueDate, and paid\_?) that are not applicable to a draft invoice. The send method now looks like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def send: SentInvoice = {    require(readyToSend\_?, "recipient and items must be specified before sending")    val now = new LocalDate    applySent(InvoiceSent(id, sentDate = now, dueDate = now.plusDays(14)))  }    private def applySent(event: InvoiceSent) = new SentInvoice(event :: uncommittedEvents, id, event.dueDate) |

There are a couple of changes here:

* The method returns a SentInvoice.
* The run-time check to see if the invoice was already sent before is removed, since this is impossible for a draft invoice.
* A new method applySent was added and is called from send instead of the generic applyEvent method. The reason is that the genericapplyEvent returns an Invoice, not a SentInvoice.

The addition of the applySent method means that the applyEvent method needs to be changed accordingly:

|  |  |
| --- | --- |
| 1  2  3  4  5 | def applyEvent = {    // [... code omitted ...]    case event: InvoiceSent => applySent(event)    // [... code omitted ...]  } |

Finally, the loadFromHistory method inherited by the Invoicecompanion object from AggregateFactory returns a plain Invoice, which is now an empty interface. Not very client friendly. Let’s change it so that the client can specify the expected type:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | trait AggregateFactory[AR <: AggregateRoot[AR, Event], Event] extends EventSourced[Event] {    def loadFromHistory[T <: AR](history: Iterable[Event]): T = {      var aggregate = applyEvent(history.head)      for (event <- history.tail)        aggregate = aggregate.applyEvent(event)      aggregate.asInstanceOf[AR].markCommitted.asInstanceOf[T]    }  } |

That’s it! The rest of the classes can be implemented in a similar fashion. Just for fun, here’s the full implementation of the PaidInvoiceclass:

|  |  |
| --- | --- |
| 1  2  3  4  5 | case class PaidInvoice(uncommittedEvents: List[InvoiceEvent]) extends Invoice {    def markCommitted = copy(uncommittedEvents = Nil)      def applyEvent = unhandled  } |

It’s now immediately clear that the *paid* state of an invoice is a final state. No domain behavior is provided and no events are accepted. There is only some boilerplate code left.

From a client perspective little has changed, except that the bad call sequence at the start of this blog will no longer compile. And if you’re using an IDE, code completion will be helped by the additional compile-time information. Another nice thing is that unit tests like these are no longer required:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | "draft invoice" should {    val invoice: DraftInvoice = Invoice.loadFromHistory(Seq(InvoiceCreated(1)))      "not be payable" in {      invoice.pay must throwA[IllegalArgumentException]    }  } |

In fact, tests like these will no longer compile at all!

The complete code of the typed invoice implementation can be found in [Invoice.scala](https://github.com/erikrozendaal/immutable-domain-example/blob/master/src/main/scala/com/zilverline/examples/immutabledomain/typed/Invoice.scala). The amount of boilerplate code has slightly increased with the addition of typed event handlers and the need to reimplement the uncommittedEvents and markCommitted methods in each Invoice subclass.

But the domain logic has been simplified and each class is now much more cohesive, with less data to manage. Also notice that the dueDatefield of a SentInvoice is no longer an Option[LocalDate], but simply aLocalDate. This is because a sent invoice *always* has a due date, and we no longer need to make the field optional just because the class also needs to support the draft invoice state.

Futhermore, methods like readyToPay\_? are no longer needed. The only remaining runtime checks are those that are truly dynamic (likelate\_?) or that are harder to encode in the type system such asreadyToSend\_?. Maybe the latter constraint can still be encoded with[phantom types](http://blog.rafaelferreira.net/2008/07/type-safe-builder-pattern-in-scala.html), but that seems overly complicated for the current problem.

**Conclusion**

The immutable representation of our domain made it easy to adapt the invoice type dynamically, based on the changes applied to the invoice. This helped us to implement the domain in a more concise and understandable fashion, while also improving type safety and reducing the need to implement runtime checks. Basically we’re getting a statically typed GoF [State Pattern](http://en.wikipedia.org/wiki/State_pattern) for free.

For me this may very well be the nicest features of immutable domain models and object-functional programming. Many, many kinds of business domain objects have a lifecycle, where behavior changes in each state. Immutable objects made it possible to directly encode this.

In the fifth (and final) part we’ll look at using the swiss army nuclear rocket of programming to reduce the amount of boilerplate code related to tracking uncommitted events: the [monad](http://www.codecommit.com/blog/ruby/monads-are-not-metaphors).

# 5. monads

This is the fifth and final part of this series. In this last part we’ll reduce the boilerplate code related to handling events and as a bonus we’ll also make handling validation a bit nicer. But before we take a deep dive into the code, let’s consider the design of the last three Invoice implementations.

**Design comparison**

Since this part is mostly about reducing boilerplate from the typed implementation of invoice without significantly affecting the design of our domain classes, it makes sense to take a quick look at the design of the three event sourced implementations so far: mutable (part 2), immutable (part 3), and typed/monadic (part 4/5). Which one, if any, is the “best”? Let’s compare them based on [XP’s rules of simple design](http://c2.com/cgi/wiki?XpSimplicityRules).

| **Rule** | **Mutable** | **Immutable** | **Typed / Monadic** |
| --- | --- | --- | --- |
| Pass all tests | ++ | ++ | ++ |
| Clear, expressive, & consistent | + | + | ++ |
| No duplication | – | – | + |
| Minimal methods, classes | + | + | -/+ |

So all implementations pass the tests and therefore satisfy the requirements. Splitting the invoice class into three subclasses improved expressiveness and reduced duplication (no need for some runtime checks), at the cost of adding some new methods and classes.

But the main highlight is that we were able to *radically redesign and re-implement* our domain code with *minimal* impact on our clients or infrastructure. We still generate exactly the same events as in the first mutable event sourced implementation and only the calling syntax was slightly changed (which usually is contained in a controller or service layer anyway). The reporting side of the application, any views, integration with external systems, etc. are unaffected!

So by having our events as a [stable abstraction](http://www.objectmentor.com/resources/articles/stability.pdf) we have greatly decoupled the components of our application while making each component *more cohesive*. This gives us confidence that as requirements change we can keep our domain clean and simple, without needing to compromise our implementation with respect to durability or reporting needs.

So now that we have this out of the way, let’s get back into the code.

**Composing event handlers**

Let’s take a look at the applyEvent method in the DraftInvoice class from part 4:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def applyEvent = {    case event: InvoiceRecipientChanged => applyRecipientChanged(event)    case event: InvoiceItemAdded => applyItemAdded(event)    case event: InvoiceItemRemoved => applyItemRemoved(event)    case event: InvoiceSent => applySent(event)    case event => unhandled(event)  } |

It’s basically checking the type of the event and then dispatching to the correct event handler using a case-block. Each event handler is simply a function that takes an event of the correct type and returns the appropriate response. We could try to use some reflection magic to remove the need for this method, but that feels like cheating. Let’s try to if we can *compose* our typed event handlers into the generic applyEvent method instead.

Fortunately, Scala has a built-in trait called [PartialFunction](http://www.scala-lang.org/api/current/scala/PartialFunction.html) that has the useful method orElse that does exactly what we need. We just need to turn our event handling functions into partial functions to make this work.

We do this by making the type of our handlers explicit using a new class EventHandler and adding an *implicit* conversion from our handler type to partial functions. The partial function checks the event’s type and invoke the handler if the type is correct:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | protected class EventHandler[Event, +Result](callback: Event => Result) {    def apply(event: Event) = // [... code omitted ...]      def applyFromHistory(event: Event) = callback(event)  }    protected def handler[A, B](callback: A => B) = new EventHandler(callback)    implicit protected def handlerToPartialFunction[A, B](handler: EventHandler[A, B])  (implicit m: Manifest[A]) =    new PartialFunction[AnyRef, B] {      def isDefinedAt(event: AnyRef) = m.erasure.isInstance(event)        def apply(event: AnyRef) = handler.applyFromHistory(event.asInstanceOf[A])    } |

Here an instance of EventHandler can be created using the handler method. The EventHandler class has an applyFromHistory method which simply passes the event to the provided callback.

The handlerToPartialFunction takes an EventHandler and an implicit [Manifest](http://stackoverflow.com/questions/3587286/how-does-scalas-2-8-manifest-work#3588191) and turns it into a partial function that takes an event of type AnyRef (Scala’s equivalent of Java’s Object class). When the partial function is invoked it downcasts the event and delegates to theEventHandler‘s applyFromHistory method. But the partial function is only defined when the provided event’s type matches the type expected by the event handler!

Using these definitions we can now update our typed event handlers and easily compose them for use with applyEvent, using the standardPartialFunction.orElse method:

|  |  |
| --- | --- |
| 1  2  3  4  5 | def applyEvent = applyRecipientChanged orElse applyItemAdded  orElse applyItemRemoved orElse applySent    private def applyRecipientChanged = handler {event: InvoiceRecipientChanged =>    copy(event :: uncommittedEvents, recipient\_? = event.recipient.isDefined)  } |

Now the applyEvent method is just a single line listing all the applicable event handlers, without all the boilerplate. The typed event handlers are slightly changed to include the call to the handler factory method.

**Extracting uncommitted events**

The other part of boilerplate code was related to storing and managing the list of uncommitted events. Each aggregate root subclass needed to provide an accessor for the uncommittedEvent field and implement the markCommitted method. Each event handler then had to ensure the new event was prepended to the list of uncommitted event, which is somewhat error-prone.

The first step to fixing this is to remove the list of uncommitted events from the aggregate root and to start explicitly passing it into our methods, which then returns the updated list together with the updated invoice. This will certainly clean up the invoice subclasses, such as the PaidInvoice listed below.

|  |  |
| --- | --- |
| 1  2  3 | class PaidInvoice extends Invoice {    def applyEvent = unhandled  } |

Unfortunately it makes methods like pay and applyPaymentReceivedrather ugly, and we’re not even talking yet about the client code which now needs to manually manage the list of uncommitted events:

|  |  |
| --- | --- |
| 1  2  3  4  5 | def pay(uncommittedEvents: List[Any]) =    applyPaymentReceived(InvoicePaymentReceived(id, new LocalDate), uncommittedEvents)    private def applyPaymentReceived(event: InvoicePaymentReceived,  uncommittedEvents: List[Any]) =    (event :: uncommittedEvents, new PaidInvoice) |

Fortunately, [monads](http://www.codecommit.com/blog/ruby/monads-are-not-metaphors) can help us with that. But first we’ll make our types explicit (a recurring theme of this series).

Let’s look at the types we have now. The pay method above has the type List[Any] => (List[Any], PaidInvoice). Let’s call the type of pay a Behaviour which returns a Reaction when triggered by passing it a list of events. A reaction can either be Accepted for success or Rejected when failed:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | trait Reaction[+T]  case class Accepted[+T](events: List[Any], result: T) extends Reaction[T]  case class Rejected(message: String) extends Reaction[Nothing]    trait Behavior[+A] {    protected def apply(events: List[Any]): Reaction[A]      // [... code omitted ...]      def reaction = apply(Nil)      def changes = reaction.asInstanceOf[Accepted[\_]].events      def rejected = reaction.asInstanceOf[Rejected].message  } |

The Behavior class defines an abstract method apply that takes the current list of uncommitted events as argument and should implement the specific behavior we want. It also adds a reaction method that invokes the behavior with the empty list. The changes and rejected methods are just there for convenience.

We’ll also define a few additional methods to create some useful behaviors:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | object Behaviors {    def behavior[T](callback: List[Any] => Reaction[T]) = new Behavior[T] {      protected def apply(events: List[Any]) = callback(events)    }      def accept[T](result: T) = behavior(events => Accepted(events, result))      def reject(message: String) = behavior(\_ => Rejected(message))      def record(event: Any) = behavior(events => Accepted(event :: events, ()))      def guard(condition: Boolean, message: => String) =  if (condition) accept() else reject(message)  } |

The behavior method lets us easily create a new behavior by providing a callback, instead of having to define a new anonymous subclass implementation every time.

Accept doesn’t modify the list of uncommitted events but simple returns a specific result, reject forgets about any uncommitted events and returns an error message, record records the provided event and returns an uninteresting value, and guard rejects when the condition does not hold and returns an uninteresting value otherwise.

Now we’ll just need two more pieces to complete the puzzle. The first thing we need to be able to do is to compose two behaviors into a single new behavior. We do this by first triggering the first behavior, and if successful, passing the result from the first behavior into the second behavior. This is the monad bind operation, which is called flatMap in Scala and we make it part of our Behavior trait:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | trait Behavior[+A] {    protected def apply(events: List[Any]): Reaction[A]      def flatMap[B](next: A => Behavior[B]) = behavior {events =>      this(events) match {        case Accepted(updatedEvents, result) => next(result)(updatedEvents)        case Rejected(message) => Rejected(message)      }    }      // [... code omitted ...]  } |

With all of this in place, we can define the EventHandler.apply method which is used by our invoice implementation to call event handlers when not reloading from history:

|  |  |
| --- | --- |
| 1  2  3  4  5 | protected class EventHandler[Event, +Result](callback: Event => Result) {    def apply(event: Event) = record(event) flatMap (\_ => accept(callback(event)))      // [... code omitted ...]  } |

The apply method simply *records* the event and then *accepts*whatever the result of the event handler’s callback is. The return value of record is ignored, since it is of type Unit and not interesting.

So let’s take a look at the DraftInvoice.send method of our new[Invoice.scala](https://github.com/erikrozendaal/immutable-domain-example/blob/master/src/main/scala/com/zilverline/examples/immutabledomain/typedmonadic/Invoice.scala):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def send: Behavior[SentInvoice] =    guard(readyToSend\_?, "recipient and items must be specified before sending")  flatMap {\_ =>      val now = new LocalDate      applySent(InvoiceSent(id, sentDate = now, dueDate = now.plusDays(14)))    }    // [... code omitted ...]    private def applySent = handler {event: InvoiceSent =>  new SentInvoice(id, event.dueDate)} |

This send method’s only changes are the return type (Behavior[SentInvoice]) and the use of the guard method to perform validation. Again flatMap is used to sequence the behavior. So if the guard fails the applySent is never performed. The implementation ofapplySent is now also cleaned up and no longer has to worry about recording the event as this is taken care of by EventHandler.apply.

Client code is now unfortunately a bit more complicated, since it needs to deal with the monad:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | "draft invoice" should {    val invoice: DraftInvoice = Invoice.loadFromHistory(Seq(InvoiceCreated(1)))      "support adding invoice items" in {  val updated = invoice.addItem("Food", "2.95")  flatMap (\_.addItem("Water", "1.95")) flatMap (\_.removeItem(1))    updated.changes must contain(InvoiceItemAdded(  1, InvoiceItem(1, "Food", "2.95"), "2.95"))  updated.changes must contain(InvoiceItemAdded(  1, InvoiceItem(2, "Water", "1.95"), "4.90"))  updated.changes must contain(InvoiceItemRemoved(  1, InvoiceItem(1, "Food", "2.95"), "1.95"))    }      "not be ready to send" in {  invoice.send.rejected must beEqualTo("recipient and  items must be specified before sending")    }  } |

Here the need to sequence the different behaviors using flatMap is ugly. Fortunately, this can be improved by using a more convenient name (bind or >>=) and in many cases, a service will only invoke a single method on an aggregate at a time, so the need for sequencing behaviors may be rare. Validation checking has improved, since there is no longer a need to catch exceptions, improving readability and making it easier to combine multiple validation results into one.

**Conclusion**

This series of blog posts has taken us through a whirlwind tour of modelling a simple invoice example. Starting out with a straightforward JPA implementation we quickly moved to event sourcing that made it possible to implement an *immutable domain model*. This allowed us to raise the level of abstraction of our code, increase clarity, and increase type safety. Finally we used some functional programming techniques to reduce boilerplate. Even with the additional classes and event handling methods, the monadic version of Invoice is only slightly larger than the mutable event sourced invoice. The test code is significantly smaller, since it no longer needs to test for various run-time checks, as the compiler takes care of that.

But the most significant advantage of event sourcing is the ability to change the implementation of the domain in radical ways, making it possible to keep up with changing business requirements and allowing us to keep improving the domain as our knowledge and understanding improves. All this with very little impact on the rest of the system and no need to perform database migrations.

Furthermore, our reporting and querying needs are also decoupled from the domain. Now we can easily use an RDBMS, document store, lucene index, graph database, and/or anything else that is most appropriate for our specific querying and reporting needs, without any impact on our domain model.

Finally, critical business data has become much more durable, as we only *add* events, and never update or delete. Full historical data is maintained, which potentially holds a great amount of business value. We now also have access to full audit logs for regulatory or debugging purposes, etc. This is incredibly valuable!

The design space for CQRS, event sourcing, and immutable domain models is still wide open. It will be very interesting to see how this evolves, and when and where these techniques are applicable. Certainly it makes it possible to apply standard functional programming techniques to “inherently” mutable business domains, with all the goodness that entails.