Jimmy Nilsson

Applying Domain-Driven Design and Patterns

With Examples in C# and .NET

Forewords by Martin Fowler and Eric Evans

Praise for *Applying Domain-Driven Design and Patterns*

"I don't know what it was I professed to doing before I had added Domain-Driven Design and Test-Driven Development to my toolkit, but from my present perspective, I'm reticent to call it anything but chaotic hacking. Domain-Driven Design and Test-Driven Development are two approaches that have consistently guided me toward a practical application of software design principles, and brought my projects to fruition with healthy, sustainable software. This is a book that puts its money where its mouth is in terms of concretely communicating Agile software development practice. It's potentially one of the most impactful guides to software development success practices yet offered to the .NET software development community."
—Scott Bellware, Microsoft MVP for C#, DDD and TDD Blogger

"Jimmy Nilsson does his readers a great service by showing them how to apply the foundational principles of enterprise application design and development taught by Evans, Fowler, and other thought leaders. The book uses a worked example not only to explain, but also to demonstrate Domain-Driven Design, Patterns of Enterprise Application Architecture, and Test-Driven Development. Jimmy's insight and experience make reading it a pleasure, and leave the reader with the certainty that they have learned from a master practitioner. Enterprise developers looking to master these principles will find the book a valuable guide."
—Jack Greenfield, Enterprise Tools architect, Visual Studio Team System, Microsoft

"Good software architects reserve the right to get smarter. Beyond the goal of shipping their current system, they're on a quest to discover better ways to design and build software. This book is travelogue of sorts in which Jimmy documents his journey through a wide range of patterns, practices, and technologies, explaining how his thinking about enterprise systems has evolved along the way. If you're traveling the same road, this book is a good companion."
—Tim Ewald, principal architect at Foliage Software Systems and author of *Transactional COM+: Building Scalable Applications*

```
    Customer newCustomer = _CreateACustomer("Ronneby");

    //Inject a stubbed version of CreditService
    //that won't allow a credit of more than 300.
    newCustomer.CreditService = new StubCreditService(300);

    newCustomer.CreditLimit = 1000;

    Assert.IsFalse(newCustomer.HasOKCreditLimit);
}
```

Note

Gregory Young pointed this out as a code smell. The problem is most likely to be that the operation isn't atomic. The value is first set (and therefore the old value is overwritten) and then checked (if remembered). Perhaps something like `Customer.RequestCreditLimit(1000)` would be a better solution. We get back to similar discussions in Chapter 7, "Let the Rules Rule."

9. An Order Must Have a Customer; an Orderline Must Have an Order

Feature 9 is a common and reasonable requirement, and I think it's simply and best dealt with by referential integrity constraints in the database. We can, and should, test this in the Domain Model, too. There's no point in sending the Domain Model changes to persistence if it has obvious incorrectness like this. But instead of checking for it, as the first try I make it kind of mandatory thanks to an `OrderFactory` class as the way of creating `Orders`, and while I'm at it I deal with `OrderLines` the same way so an `OrderLine` must have `Order` and `Product` (see Figure 4-11).

Let's take a look at the interaction between the different parts in a test as usual.

```
[Test]
public void CanCreateOrderWithOrderLine()
{
    Customer newCustomer = _CreateACustomer("Karlskrona");

    Order newOrder = OrderFactory.CreateOrder(newCustomer);

    //The OrderFactory will use AddOrderLine() of the order.
    OrderFactory.CreateOrderLine(newOrder, new Product());

    Assert.AreEqual(1, newOrder.OrderLines);
}
```

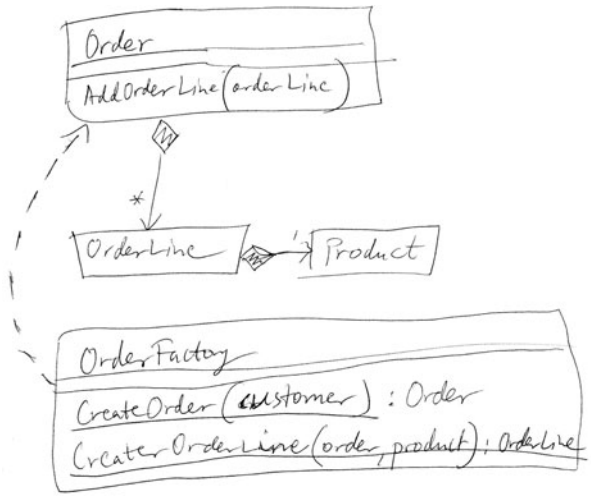

Figure 4-11 OrderFactory and enhanced Order

Hmmm… This feels quite a bit like overdesign and not exactly fluent and smooth, either. Let's see what we think about it when we really get going in the next chapter.

While we are discussing the OrderFactory, I want to mention that I like the idea of using Null Objects [Woolf Null Object] for OrderType, Status, and ReferencePerson. (That goes both for the Domain Model and actually also for the underlying relational database.) The Null Object pattern means that instead of using null, you use an empty instance (where empty means default values for the members, such as string.Empty for the strings). That way you can always be sure to be able to "follow the dots" like this:

```
this.NoNulls.At.All.Here.Description
```

When it comes to the database, you can cut down on outer joins and use inner joins more often because foreign keys will at least point to the null symbols and the foreign key columns will be non-nullable. To summarize, null objects increase the simplicity a lot and in unexpected ways as well.

As you saw in Figure 4-11, I had also added a method to the Order class, AddOrderLine(), for adding OrderLines to the Order. That's part of an implementation of the Encapsulate Collection Refactoring [Fowler R], which basically means that the parent will protect all changes to the collection.

On the other hand, the database is the last outpost, and I think this rule fits well there, too.

10. Saving an Order and Its Lines Should Be Atomic

Again, I see `Order` and its `OrderLines` as an Aggregate, and the solution I plan to use for this feature will be oriented around that.

I will probably use an implementation of the Unit of Work pattern [Fowler PoEAA] for keeping track of the instances that have been changed, which are new, and which are deleted. Then the Unit of Work will coordinate those changes and use one physical database transaction during persistence.

11. Orders Have an Acceptance Status

As we specified, orders have an acceptance status (see Figure 4-12). Therefore, I just add a method called `Accept()`. I leave the decision about whether or not to internally use the State pattern [GoF Design Patterns] for later. It is better to make implementation decisions like this during refactoring.

Figure 4-12 `Order` and `Status`

When we discuss other states for `Order`, we add more methods. For now, nothing else has been explicitly required so we stay with just `Accept()`.

The current idea could look like this:

```
[Test]
public void CanAcceptOrder()
{
    Customer newCustomer = _CreateACustomer("Karlskrona");
    Order newOrder = OrderFactory.CreateOrder(newCustomer);

    Assert.IsFalse(newOrder.Status == OrderStatus.Accepted);

    newOrder.Accept();

    Assert.IsTrue(newOrder.Status == OrderStatus.Accepted);
}
```

The Domain Model to This Point

So if we summarize the Domain Model just discussed, it could look like Figure 4-13.

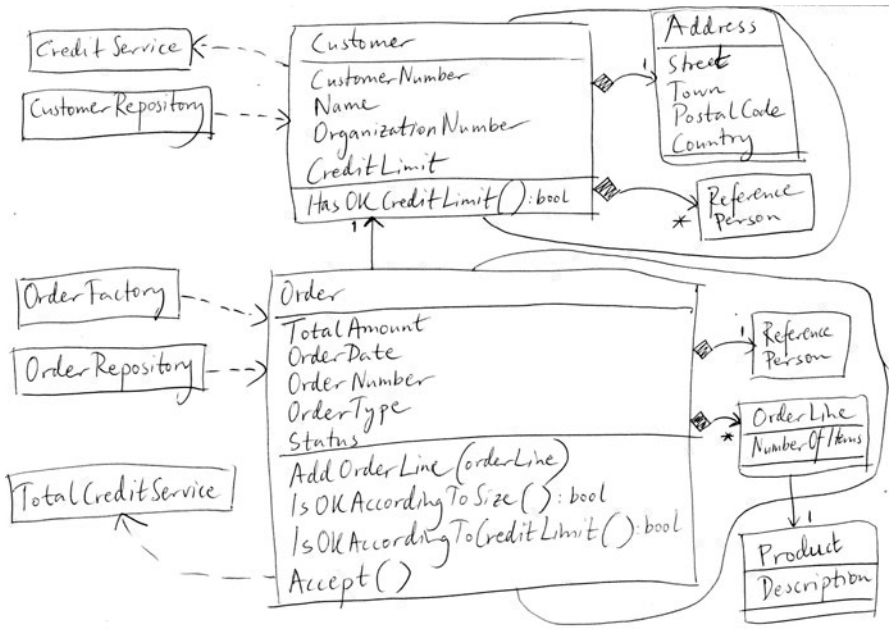

Figure 4-13 A sketched Domain Model for how I now think I will approach the feature list

Note

The class `ReferencePerson` is in two different Aggregates in Figure 4-13, but the instances aren't. That's an example of how the static class diagram lacks in expressiveness, but also an example that is simply explained with a short comment.

What's that? Messy? OK, I agree, it is. We'll partition the model better when we dive into the details further.

Note

The model shown in Figure 4-13 and all the model fragments were created up-front, and they are bound to be improved a lot when I move on to developing the application.

> Another note is that I only showed the core Domain Model and not things that are more infrastructure related, such as the mentioned Unit of Work [Fowler PoEAA]. Of course, that was totally on purpose. We'll get back to infrastructure later in the book. Now I'm focusing on the Domain Model.

I said that there are lots of variations of how the Domain Model pattern is used. To show you some other styles, I asked a couple of friends of mine to describe their favorite ways of applying Domain Models. You'll find those in Appendix A.

To give us a better feeling of the requirements, I'd like to take a look at them from yet another view and sketch a few forms for an upcoming UI.

Making a First Attempt at Hooking the UI to the Domain Model

I once started describing a new architecture from the bottom up. I mean, I started describing it from the database. One of the readers told me in no uncertain terms that I should start talking about what the architecture looked like from the UI programmer's point of view. If it wasn't good in that perspective, there was no point reading on. I decided that that way of thinking had its merits, so it's now time to have a first look at the Domain Model we just sketched from a UI programmer's viewpoint.

A Basic Goal

I want you to consider whether you think we are fulfilling one of my basic goals, which is providing "simplicity to the left" (or "simplicity to the top," depending upon how you visualize the layers in a layered architecture). What I mean is providing a simple API for the UI programmer so he or she can easily see, breathe, and understand the model, and can focus on UI matters and not have to think about complex protocols for the Domain Model.

Skipping data binding is not a basic goal, not at all, but I will skip data binding here in the first discussion about the UI. Data binding won't be the focus at all in this book, but we'll touch some more on that in later chapters.

The Current Focus of the Simple UI

I think the Domain Model might be somewhat abstract at the moment, but discussing it from a UI point of view might change this a bit. The scenarios I think we should try out are the following:

- List orders for a customer
- Add an order

Again, I won't use ordinary databinding right here, but just simple, direct code for hooking the UI to the Domain Model.

List Orders for a Customer

I need to be able to list orders for a customer. When the Domain Model is built later on, I just need to add a "view" on top of the Domain Model and fake some `Orders`. I'm thinking about a form that looks something like the one in Figure 4-14.

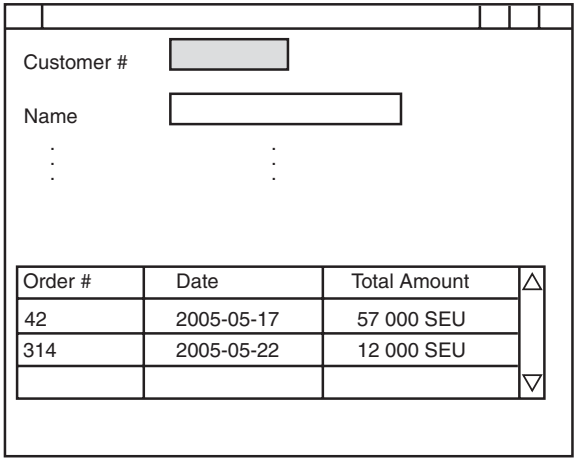

Figure 4-14 List orders for a customer

Note

You might think that I'm fighting the tool (VS.NET or similar form editors), but because I'm in extremely early sketch mode, I decided to visualize the form ideas with pen and paper. (But for this book, I'll provide images that were rendered in a graphics program.)