AUGUST 2013

VOLUME 28 NUMBER 8

Data Points - Coding for Domain-Driven Design: Tips for Data-Focused Devs

By [Julie Lerman](https://msdn.microsoft.com/en-us/magazine/mt149362?author=julie+lerman) | August 2013 | Get the Code: [C#](http://download.microsoft.com/download/C/4/0/C40DE96D-55C4-4911-83D2-7C880A46C2B0/Code_LermanDataPoints0813.zip)   [VB](http://download.microsoft.com/download/C/4/0/C40DE96D-55C4-4911-83D2-7C880A46C2B0/Code_LermanDataPoints0813.zip)

This year, Eric Evans’ groundbreaking software design book, “Domain-Driven Design: Tackling Complexity in the Heart of Software” (Addison-Wesley Professional, 2003, [amzn.to/ffL1k](http://amzn.to/ffL1k)), celebrates its 10th anniversary. Evans brought to this book many years of experience guiding large businesses through the process of building software. He then spent more years thinking about how to encapsulate the patterns that lead these projects to success—interacting with the client, analyzing the business problems being solved, building teams and architecting the software. The focus of these patterns is the business’s domain, and together they comprise Domain-Driven Design (DDD). With DDD, you model the domain in question. The patterns result from this abstraction of your knowledge about the domain. Rereading Martin Fowler’s foreword and Evans’ preface even today, continues to provide a rich overview of the essence of DDD.

In this column and the next two as well, I’ll share some pointers that have helped my data-focused, Entity Framework brain gain clarity as I work on getting my code to benefit from some DDD technical patterns.

**Why Do I Care About DDD?**

My introduction to DDD came from a short video interview on [InfoQ.com](http://infoq.com/) with Jimmy Nilsson, a respected architect in the .NET community (and elsewhere), who was talking about LINQ to SQL and the Entity Framework ([bit.ly/11DdZue](http://bit.ly/11DdZue)). At the end, Nilsson is asked to name his favorite tech book. His reply: “My favorite computer book is the book by Eric Evans, “Domain-Driven Design.” It’s like poetry, I think. It’s not just great content, but you can read it many times and it reads like poetry.” Poetry! I was writing my first tech book, “Programming Entity Framework” (O’Reilly Media, 2009), at the time, and I was intrigued by this description. So I went and read a little bit of Evans’ book to see what it was like. Evans is a beautiful, fluid writer. And that, combined with his perceptive, naturalistic view of software development, does make the book a joy to read. But I was also surprised by what I was reading. Not only was the writing wonderful, what he was writing about intrigued me. He talked about building relationships with clients and truly understanding their businesses and their business problems (related to the software in question), not just slogging code. This is something that’s been important to me in my 25 years of software development. I wanted more.

I tiptoed around the edge of DDD for a few more years, then started learning more—meeting Evans at a conference and then attending his four-day immersion workshop. While I’m far from an expert in DDD, I found that the Bounded Context pattern was something I could leverage right away as I worked to shift my own software creation process toward a more organized, manageable structure. You can read about that topic in my January 2013 column, “Shrink EF Models with DDD Bounded Contexts” ([msdn.microsoft.com/magazine/jj883952](https://msdn.microsoft.com/magazine/jj883952)).

Since then I’ve explored further. I’m intrigued and inspired by DDD, but struggle with my data-driven perspective to comprehend some of the technical patterns that make it successful. It seems likely that many developers go through the same struggle, so I’m going to share some of the lessons I’ve been learning with the help, interest, and generosity of Evans and a number of other DDD practitioners and teachers, including Paul Rayner, Vaughn Vernon, Greg Young, Cesar de la Torre, and Yves Reynhout.

**When Modeling the Domain, Forget About Persistence**

Modeling the domain is all about focusing on the tasks of the business. When designing types and their properties and behaviors, I’m sorely tempted to think about how a relationship will work out in the database and how my object relational mapping (ORM) framework of choice—Entity Framework—will treat the properties, relationships and inheritance hierarchies that I’m building. Unless you’re building software for a company whose business is data storage and retrieval—something like Dropbox—data persistence only plays a supporting role in your application. It’s much like making a call out to a weather source’s API in order to display the current temperature to a user. Or sending data from your app to an external service, perhaps a registration on Meetup.com. Of course, your data may be more complicated, but with a DDD approach to bounding contexts, focusing on behaviors and following DDD guidance when building types, the persistence can be much less complex than the systems you may be building today.

And if you’ve studied up on your ORM, such as learning how to configure database mappings using the Entity Framework Fluent API, you should be able to make the persistence work as needed. In the worst case, you may need to make some tweaks to your classes. In an extreme case, such as with a legacy database, you could even add in a persistence model designed for database mapping, then use something such as AutoMapper to resolve things between your domain model and your persistence model.

But these concerns are unrelated to the business problem your software is aimed at solving, so persistence should not interfere with the domain design. This is a challenge for me because as I’m designing my entities, I can’t help but consider how EF will infer their database mappings. And so I try to block out that noise.

**Private Setters and Public Methods**

Another rule of thumb is to make property setters private. Instead of allowing calling code to randomly set various properties, you should control interaction with DDD objects and their related data using methods that modify the properties. And, no, I don’t mean methods like SetFirstName and SetLastName. For example, instead of instantiating a new Customer type and then setting each of its properties, you might have some rules to consider when creating a new customer. You can build those rules into the Customer’s constructor, use a Factory Pattern method or even have a Create method in the Customer type. **Figure 1** shows a Customer type that’s defined following the DDD pattern of an aggregate root (that is, the “parent” of a graph of objects, also referred to as a “root entity” in DDD). Customer properties have private setters so that only other members of the Customer class can directly affect those properties. The class exposes a constructor to control how it’s instantiated and hides the parameter-less constructor (required by Entity Framework) as internal.

**Figure 1 Properties and Methods of a Type That Acts As an Aggregate Root**

public class Customer : Contact

{

  public Customer(string firstName,string lastName, string email)

  { ... }

  internal Customer(){ ... }

  public void CopyBillingAddressToShippingAddress(){ ... }

  public void CreateNewShippingAddress(

   string street, string city, string zip) { ... }

  public void CreateBillingInformation(

   string street, string city, string zip,

   string creditcardNumber, string bankName){ ... }

  public void SetCustomerContactDetails(

   string email, string phone, string companyName){ ... }

  public string SalesPersonId { get; private set; }

  public CustomerStatus Status{get;private set;}

  public Address ShippingAddress { get; private set; }

  public Address BillingAddress { get;private set; }

  public CustomerCreditCard CreditCard { get; private set; }

}

The Customer type controls and protects the other entities in the aggregate—some addresses and a credit-card type—by exposing specific methods (such as CopyBillingAddressToShippingAddress) with which those objects will be created and manipulated. The aggregate root must make sure the rules that define each entity within the aggregate are applied using domain logic and behavior implemented in these methods. Most important, the aggregate root is in charge of invariant logic and consistency throughout the aggregate. I’ll talk more about invariants in my next column, but for the meantime, I recommend reading Jimmy Bogard’s blog post, “Strengthening Your Domain: Aggregate Construction,” at [bit.ly/ewNZ52](http://bit.ly/ewNZ52), which provides an excellent explanation of invariants in aggregates.

In the end, what’s exposed by Customer is behavior rather than properties: CopyBillingAddressToShippingAddress, CreateNewShipping­Address, CreateBillingInformation and SetCustomerContactDetails.

Note that the Contact type, from which Customer derives, lives in a different assembly named “Common” because it may be needed by other classes. I need to hide the properties of Contact, but they can’t be private or Customer wouldn’t be able to access them. Instead, they’re scoped as Protected:

public class Contact: Identity

{

  public string CompanyName { get; protected set; }

  public string EmailAddress { get; protected set; }

  public string Phone { get; protected set; }

}

A side note about Identities: Customer and Contact may look like DDD value objects because they have no key value. However, in my solution, the key value is provided by the Identity class from which Contact derives. And neither of these types are immutable, so they can’t be considered value objects anyway.

Because Customer inherits from Contact, it will have access to those protected properties and is able to set them, as in this SetCustomerContactDetails method:

public void SetCustomerContactDetails  (string email, string phone, string companyName)

{

  EmailAddress = email;

  Phone = phone;

  CompanyName = companyName;

}

**Sometimes All You Need Is CRUD**

Not everything in your app needs to be created using DDD. DDD is there to help handle complex behaviors. If you just need to do some raw, random editing or querying, then a simple class (or set of classes), defined just as you’d typically do with EF Code First (using properties and relationships) and combined with insert, update and delete methods (via a repository or just DbContext), is all you need. So, to accomplish something like creating an order and its line items, you might want DDD to help work through special business rules and behaviors. For example, is this a Gold Star customer placing the order? In that case, you need to get some customer details to determine if the answer is yes, and, if so, apply a 10 percent discount to each item being added to the order. Has the user provided their credit-card information? Then you might need to call out to a verification service to ensure it’s a valid card.

The key in DDD is to include the domain logic as methods within the domain’s entity classes, taking advantage of OOP instead of implementing “transactional scripts” within stateless business objects, which is what a typical demo-ware Code First class looks like.

But sometimes all you’re doing is something very standard, like creating a contact record: name, address, referred by, and so forth, and saving it.  That’s just create, read, update and delete (CRUD). You don’t need to create aggregates and roots and behaviors to satisfy that.

Most likely your application will contain a combination of complex behaviors and simple CRUD. Take the time to clarify the behaviors and don’t waste time, energy and money over-architecting the pieces of your app that are really just simple. In these cases, it’s important to identify boundaries between different subsystems or bounded contexts. One bounded context could be very much data-driven (just CRUD), while a critical core-domain-bounded context should, on the other hand, be designed following DDD approaches.

**Shared Data Can Be a Curse in Complex Systems**

Another issue I banged my head on, then ranted and whined about as people kindly tried to explain further, concerned sharing types and data across subsystems. It became clear that I couldn’t “have my cake and eat it too,” so I was forced to think again about my assumption that I absolutely positively must share types across systems and have those types all interact with the same table in the same database.

I’m learning to really consider where I need to share data, and then pick my battles. Some things just may not be worth trying, like mapping from different contexts to a single table or even a single database. The most common example is sharing a Contact that’s trying to satisfy everyone’s needs across systems. How do you reconcile and leverage source control for a Contact type that might be needed in numerous systems? What if one system needs to modify the definition of that Contact type? With respect to an ORM, how do you map a Contact that’s used across systems to a single table in a single database?

DDD guides you away from sharing domain models and data by explaining that you don’t always need to point to the same person table in a single database.

My biggest push back with this is based on 25 years of focusing on the benefits of reuse—reusing code and reusing data. So, I have a hard time with the following idea but I’m warming up to it: It isn’t a crime to duplicate data. Not all data will fit into this new (to me) paradigm, of course. But what about something lightweight like a person’s name? So what if you duplicate a person’s first and last name in multiple tables or even multiple databases that are dedicated to different sub-systems of your software solution? In the long run, by letting go of the complexity of sharing data, you make the job of building your system much simpler. In any case, you must always minimize data and attribute duplication in different bounded contexts. Sometimes you just need the customer’s ID and status in order to calculate discounts in a Pricing-bounded context. That same customer’s first and last name might be needed only in the Contact Management-bounded context.

But there’s still so much information that needs to be shared between systems. You can leverage what DDD refers to as an “anti-­corruption layer” (which can be something as simple as a service or a message queue) to ensure that, for example, if someone creates a new contact in one system, you either recognize that the person already exists elsewhere, or ensure that the person, along with a common identity key, is created in another subsystem.

**Plenty to Chew on Until Next Month**

As I make my way through learning and comprehending the technical side of Domain-Driven Design, struggling to reconcile old habits with new ideas, and arriving at innumerable “aha!” moments, the  pointers I discussed here are truly ones that helped me see more light than darkness. Sometimes it’s just a matter of perspective, and the way I have expressed them here reflects the perspective that helped make things clearer to me.

I’ll share some more of my “aha!” moments in my next column, where I’ll talk about that condescending term you may have heard: “anemic domain model,” along with its DDD cousin, the “rich domain model.” I’ll also discuss unidirectional relationships and what to expect when it’s time to add in data persistence if you’re using Entity Framework. I’ll also touch on some more DDD topics that caused me plenty of grief in an effort to shorten your own learning curve.

Until then, why not take a closer look at your own classes and see how to be more of a control freak, hiding those property setters and exposing more descriptive and explicit methods. And, remember: no “SetLastName” methods allowed. That’s cheating!

This month’s column will continue celebrating the 10th anniversary of Eric Evans’ book, “Domain-Driven Design: Tackling Complexity in the Heart of Software” (Addison-Wesley Professional, 2003). I’ll share more tips for data-first developers who are interested in benefiting from some of the coding patterns of Domain-Driven Design (DDD). Last month’s column highlighted:

* Setting persistence concerns aside when modeling the domain
* Exposing methods to drive your entities and aggregates, but not your property setters
* Recognizing that some subsystems are perfect for simple create, read, update and delete (CRUD) interaction and don’t require domain modeling
* The benefit of not attempting to share data or types across bounded contexts

In this column you’ll learn what’s meant by the terms “anemic” and “rich” domain models, as well as the concept of value objects. Value objects seem to be a topic that falls into one of two camps for developers. It’s either so obvious that some readers can’t imagine why it warrants any discussion at all, or so baffling that others have never been able to wrap their heads around it. I’ll also encourage you to consider using value objects in place of related objects in certain scenarios.

## That Condescending Term: Anemic Domain Models

There’s a pair of terms you’ll often hear with respect to how classes are defined in DDD—anemic domain model and rich domain model. In DDD, domain model refers to a class. A rich domain model is one that aligns with the DDD approach—a class (or type) that’s defined with behaviors, not just with getters and setters. In contrast, an anemic domain model consists simply of getters and setters (and maybe a few simple methods), which is fine for many scenarios, but gains no benefit from DDD.

When I started using Entity Framework (EF) Code First, the classes I intended Code First to work with were probably 99 percent anemic. **Figure 1** is a perfect example, showing a Customer type and the Person type from which it inherits. I’d often include a handful of methods, but the basic type was merely a schema with getters and setters.

**Figure 1 Typical Anemic Domain Model Classes Look Like Database Tables**

public class Customer : Person

{

  public Customer()

  {

    Orders = new List<Order>();

  }

  public ICollection<Order> Orders { get; set; }

  public string SalesPersonId { get; set; }

  public ShippingAddress ShippingAddress { get; set; }

}

public abstract class Person

{

  public int Id { get; set; }

  public string Title { get; set; }

  public string FirstName { get; set; }

  public string LastName { get; set; }

  public string CompanyName { get; set; }

  public string EmailAddress { get; set; }

  public string Phone { get; set; }

}

I finally looked at these classes and realized I was doing little more than defining a database table in my class. It’s not necessarily a bad thing, depending on what you plan to do with the type.

Compare that to the richer Customer type listed in **Figure 2**, which is similar to the Customer type I explored in my previous column ([msdn.microsoft.com/magazine/dn342868](https://msdn.microsoft.com/magazine/dn342868)). It exposes methods that control access to properties and other types that are part of the aggregate. I’ve tweaked the Customer type since you saw it in the earlier column to better express some of the topics I’m discussing this month.

**Figure 2 A Customer Type That’s a Rich Domain Model, Not Simply Properties**

public class Customer : Contact

{

  public Customer(string firstName, string lastName, string email)

  {

    FullName = new FullName(firstName, lastName);

    EmailAddress = email;

    Status = CustomerStatus.Silver;

  }

  internal Customer()

  {

  }

  public void UseBillingAddressForShippingAddress()

  {

    ShippingAddress = new Address(

      BillingAddress.Street1, BillingAddress.Street2,

      BillingAddress.City, BillingAddress.Region,

      BillingAddress.Country, BillingAddress.PostalCode);

  }

  public void CreateNewShippingAddress(string street1, string street2,

   string city, string region, string country, string postalCode)

  {

    BillingAddress = new Address(

      street1,street2,

      city,region,

      country,postalCode)

  }

  public void CreateBillingInformation(string street1,string street2,

   string city,string region,string country, string postalCode,

   string creditcardNumber, string bankName)

  {

    BillingAddress = new Address      (street1,street2, city,region,country,postalCode );

    CreditCard = new CustomerCreditCard (bankName, creditcardNumber );

  }

  public void SetCustomerContactDetails

   (string email, string phone, string companyName)

  {

    EmailAddress = email;

    Phone = phone;

    CompanyName = companyName;

  }

  public string SalesPersonId { get; private set; }

  public CustomerStatus Status { get; private set; }

  public Address ShippingAddress { get; private set; }

  public Address BillingAddress { get; private set; }

  public CustomerCreditCard CreditCard { get; private set; }

}

In this richer model, rather than simply exposing properties to be read and written to, the public surface of Customer is made up of explicit methods. Review the Private Setters and Public Methods section of last month’s column for more details. My point here is to help you better comprehend the difference between what DDD refers to as an anemic versus a rich domain model.

## Value Objects Can Be Confusing

Though they seem simple, DDD value objects are a serious point of confusion for many, including me. I’ve read and heard so many different ways of describing value objects from a variety of perspectives. Luckily, each of the different explanations, rather than conflicting with one another, helped me build a deeper understanding of value objects.

At its core, a value object is a class that doesn’t have an identity key.

Entity Framework has a concept of “complex types” that comes fairly close. Complex types don’t have an identity key—they’re a way of encapsulating a set of properties. EF understands how to handle these objects when it comes to data persistence. In the database, they get stored as fields of the table to which the entity is mapped.

For example, instead of having a FirstName property and a LastName property in your Person class, you could have a FullName property:

public FullName FullName { get; set; }

FullName can be a class that encapsulates the FirstName and LastName properties, with a constructor that forces you to provide both of those properties.

But a value object is more than a complex type. In DDD, there are three defining characteristics of a value object:

1. It has no identity key (this aligns with a complex type)
2. It’s immutable
3. When checking its equality to other instances of the same type, all of its values are compared

Immutability is interesting. Without an identity key, a type’s immutability defines its identity. You can always identify an instance by the combination of its properties. Because it’s immutable, none of the type’s properties can change, so it’s safe to use the type’s values to identify a particular instance. (You’ve already seen how DDD entities protect themselves from random modifications by making their setters private, but you could have a method that allows you to affect those properties.) Not only does a value object hide the setters, but it prevents you from modifying any of the properties. If you modified a property, that would mean the value of the object has changed. Because the entire object represents its value, you don’t interact with its properties individually. If you need the object to have different values, you create a new instance of the object to hold a new set of values. In the end, there’s no such thing as changing a property of a value object—which effectively makes even formulating a sentence that includes the phrase “if you need to change a value of a property” an oxymoron. A useful parallel is to consider a string, which is another immutable type (at least in all languages with which I’m familiar). When working with a string instance, you don’t replace individual characters of the string. You simply create a new string.

My FullName value object is shown in **Figure 3**. It has no identity key property. You can see that the only way to instantiate it is by providing both property values, and there’s no way to modify either of those properties. So it fulfills the immutability requirement. The final requirement, that it provides a way to compare equality to another instance of that type, is hidden in the custom ValueObject class (that I’ve borrowed from Jimmy Bogard at [bit.ly/13SWd9h](http://bit.ly/13SWd9h)) from which it inherits, because that’s a bunch of convoluted code. Although this ValueObject doesn’t account for collection properties, it meets my needs because I don’t have any collections within this value object.

**Figure 3 The FullName Value Object**

public class FullName:ValueObject<FullName>

{

  public FullName(string firstName, string lastName)

  {

    FirstName = firstName;

    LastName = lastName;

  }

  public FullName(FullName fullName)

    : this(fullName.FirstName, fullName.LastName)

  {    }

  internal FullName() { }

  public string FirstName { get; private set; }

  public string LastName { get; private set; }

 // More methods, properties for display formatting

}

Keeping in mind that you may want a modified copy—for example, similar to this FullName but with a different FirstName—Vaughn Vernon, author of “Implementing Domain-Driven Design” (Addison-Wesley Professional, 2013), suggests FullName could include methods to create new instances from the existing instance:

public FullName WithChangedFirstName(string firstName)

{

  return new FullName(firstName, this.LastName);

}

public FullName WithChangedLastName(string lastName)

{

  return new FullName(this.FirstName, lastName);

}

When I add in my persistence layer (Entity Framework), it will see this as an EF ComplexType property of any class in which it’s used. When I use FullName as a property of my Person type, EF will store the properties of FullName in the database table where the Person type is stored. By default, the properties will be named FullName\_FirstName and FullName\_LastName in the database.

FullName is pretty straightforward. Even if you were doing database-focused design, you might not want to store FirstName and LastName in a separate table.

## Value Objects or Related Objects?

But now consider another scenario—imagine a Customer with a ShippingAddress and a BillingAddress:

public Address ShippingAddress { get; private set; }

public Address BillingAddress { get; private set; }

By (data-driven brain) default, I create Address as an Entity, including an AddressId property. And, again, because I “think in data,” I presume that Address will be stored as a separate table in the database. OK, I’m working hard to train myself to stop considering the database while modeling my domain, so that’s of no consequence. However, at some point I’ll add in my data layer using EF and EF will also make this assumption. But EF won’t be able to work out the mappings correctly. It will presume a 0..1:\* relationship between Address and Customer; in other words, that an address might have any number of related customers and a Customer will have either no addresses or one address.

My DBA might not like this result and, what’s more important, I won’t be writing my application code based on the same assumption that Entity Framework is making—that there are many Customers to zero or one Address. Because of this, EF might affect my data persistence or retrieval in some unexpected ways. So from the perspective of someone who’s been working with EF quite a lot, my first approach is to fix the EF mappings using the Fluent API. But asking EF to fix this problem would force me to add navigation properties from Address pointing back to Customer, which I don’t want in my model. As you can see, solving this problem with EF mappings would lead me down a rabbit hole.

When I step back and focus on the domain, not on EF or the database, it makes more sense to simply make Address a value object instead of an entity. Three steps are required:

1. I need to remove the key property from the Address type (possibly called Id or AddressId).
2. I need to be sure that Address is immutable. Its constructor already lets me populate all of the fields. I have to remove any methods that would allow any properties to change.
3. I need to ensure that Address can be checked for equality based on the values of its properties and fields. I can do that by inheriting again from that nice ValueObject class I borrowed from Jimmy Bogard (see **Figure 4**).

**Figure 4 Address as a Value Object**

public class Address:ValueObject<Address>

{

  public Address(string street1, string street2,

string city, string region,

    string country, string postalCode) {

    Street1 = street1;

    Street2 = street2;

    City = city;

    Region = region;

    Country = country;

    PostalCode = postalCode;  }

  internal Address()  {  }

  public string Street1 { get; private set; }

  public string Street2 { get; private set; }

  public string City { get; private set; }

  public string Region { get; private set; }

  public string Country { get; private set; }

  public string PostalCode { get; private set; }

  // ...

}

How I use this Address class from my Customer class won’t change, except that if I need to modify the address, I’ll have to create a new instance.

But now I no longer need to worry about managing that relationship. And it also provides another litmus test for value objects, in that a Customer really is defined by its shipping and billing information because if I sell anything to that customer, I’ll most likely need to ship it and the Accounts Payable department requires that I have its billing address.

This is not to say that every 1:1 or 1:0..1 relationship can be replaced with value objects, but it’s a nice solution here and my job became a lot simpler when I stopped having to solve one puzzle after another that popped up as I tried to force Entity Framework to maintain this relationship for me.

Like the FullName example, changing Address to be a value object means Entity Framework will see Address as a ComplexType and will store all of that data within the Customer tables. Years of being focused on database normalization made me automatically think this was a bad thing, but my database can handle it easily and it works for my particular domain.

I can think of many arguments against using this technique, but they all begin with “what if.” Any of those that don’t pertain to my domain aren’t valid arguments. We waste a lot of time coding for just-in-case scenarios that never come to pass. I’m trying to be more considerate about adding those preemptive Band-Aids into my solutions.

## Not Done Quite Yet

I’m really getting through the list of DDD concepts that have haunted this data geek. As I grok more of these concepts, I become inspired to learn even more. Once I shift my thinking a bit, these patterns make a lot of sense for me. They don’t reduce my interest in data persistence one bit, but separating the domain from the data persistence and infra­structure feels right to me after having them tangled up together for so many years. Still, it’s important to keep in mind that there are so many software activities that don’t need DDD. DDD can help sort out complex problems, but quite often it’s overkill for simple ones.

In the next column I’ll discuss a few other DDD technical strategies that also initially seemed to conflict with my data-first thinking, such as letting go of bidirectional relationships, considering invariants and dealing with any perceived need to trigger data access from your aggregates.

This is the final installment of my series on helping data-focused developers wrap their heads around some of the more challenging coding concepts used with Domain-Driven Design (DDD). As a Microsoft .NET Framework developer using Entity Framework (EF), and with a long history of data-first (and even database-first) development, I’ve struggled and argued and whined my way to understanding how to merge my skills with some of the implementation techniques of DDD. Even if I’m not using a full DDD implementation (from client interaction all the way to the code) in a project, I still benefit greatly from many of the tools of DDD.

In this last installment, I’ll discuss two important technical patterns of DDD coding and how they apply to the object-relational mapping (ORM) tool I use, EF. In an earlier installment I talked about one-to-one relationships. Here, I’ll explore unidirectional relationships—preferred with DDD—and how they affect your application. This choice leads to a difficult decision: recognizing when you might be better off without some of the nice relationship “magic” that EF performs. I’ll also talk a bit about the importance of balancing tasks between an aggregate root and a repository.

## Build Unidirectional Relationships from the Root

From the time I started building models with EF, two-way relationships have been the norm, and I did this without thinking too hard about it. It makes sense to be able to navigate in two directions. If you have orders and customers, it’s nice to be able to see the orders for a customer and, given an order, it’s convenient to access the customer data. Without thinking, I also built a two-way relationship between orders and their line items. A relationship from order to line items makes sense. But if you stop to consider this for a moment, the scenarios in which you have a line item and need to get back to its order are few and far between. One that I can think of is that you’re reporting on products and want to do some analysis on what products are commonly ordered together, or an analysis that involves customer or shipping data. In such cases, you might need to navigate from a product to the line items in which it’s contained and then back to the order. However, I can see this coming up only in a reporting scenario where I’m not likely needing to work with the DDD-focused objects.

If I only need to navigate from order to line items, what is the most efficient way to describe such a relationship in my model?

As I noted, DDD prefers unidirectional relationships. Eric Evans advises that “it’s important to constrain relationships as much as possible,” and that “understanding the domain may reveal natural directional bias.” Managing the complexities of relationships—especially when you’re depending on the Entity Framework to maintain associations—is definitely an area that can cause a lot of confusion. I’ve already penned a number of Data Points columns devoted to associations in Entity Framework. Any level of complexity that can be removed is probably beneficial.

Contemplating the simple sales model I’ve used for this series on DDD, it does present a bias in the direction of an order to its line items. I can’t imagine creating, deleting or editing a line item without starting from the order.

If you look back at the Order aggregate I built earlier in the series, the order does control the line items. For example, you need to use the CreateLineItem method of the Order class to add a new line item:

public void CreateLineItem(Product product, int quantity)

{

  var item = new LineItem

  {

    OrderQty = quantity,

    ProductId = product.ProductId,

    UnitPrice = product.ListPrice,

    UnitPriceDiscount = CustomerDiscount + PromoDiscount

  };

  LineItems.Add(item);

}

The LineItem type has an OrderId property, but no Order property. That means it’s possible to set the value of OrderId, but you can’t navigate from a LineItem to an actual Order instance.

In this case, I have, in Evans’ words, “imposed a traversal direction.” I have, in effect, ensured I can traverse from Order to LineItem but not in the other direction.

There are implications to this approach not only in the model but also in the data layer. I use Entity Framework as my ORM tool and it comprehends this relationship well enough simply from the LineItems property of the Order class. And because I happen to follow the conventions of EF, it understands that LineItem.OrderId is my foreign key property back to the Order class. If I used a different name for OrderId, things would be more complicated for Entity Framework.

But in this scenario, I can add a new LineItem to an existing order like this:

order.CreateLineItem(aProductInstance, 2);

var repo = new SimpleOrderRepository();

repo.AddAndUpdateLineItemsForExistingOrder(order);

repo.Save();

The order variable now represents a graph with a preexisting order and a single new LineItem. That preexisting order has come from the database and already has a value in OrderId, but the new LineItem has only the default value for its OrderId property, and that’s 0.

My repository method takes that order graph, adds it to my EF context and then applies the proper state, as shown in **Figure 1**.

**Figure 1 Applying State to an Order Graph**

public void AddAndUpdateLineItemsForExistingOrder(Order order)

{

\_context.Orders.Add(order);

\_context.Entry(order).State = EntityState.Unchanged;

foreach (var item in order.LineItems)

{

  // Existing items from database have an Id & are being modified, not added

  if (item.LineItemId > 0)

  {

    \_context.Entry(item).State = EntityState.Modified;

  }

}

}

In case you aren’t familiar with EF behavior, the Add method causes the context to begin tracking everything in the graph (the order and the single line item). At the same time, each object in the graph is flagged with the Added state. But because this method is focused on using a preexisting order, I know that Order is not new and, therefore, the method fixes the state of the Order instance by setting it to Unchanged. It also checks for any preexisting LineItems and sets their state to Modified so they’ll be updated in the database rather than inserted as new. In a more fleshed-out application, I’d use a pattern for more definitively knowing the state of each object, but I don’t want this sample to get bogged down with additional details. (You can see an early version of this pattern on Rowan Miller’s blog at[bit.ly/1cLoo14](http://bit.ly/1cLoo14), and an updated example in our coauthored book “Programming Entity Framework: DbContext” [O’Reilly Media, 2012].)

Because all of these actions are being done while the context is tracking the objects, Entity Framework also “magically” fixes the value of the OrderId in my new LineItem instance. Therefore, by the time I call Save, the LineItem knows that the OrderId value is 1.

## Letting Go of the EF Relationship-Management Magic—for Updates

This good fortune occurs because my LineItem type happens to follow EF convention with the foreign key name. If you named it something other than OrderId, such as OrderFK, you’d have to make some changes to your type (for example, introducing the unwanted Order navigation property) and then specify EF mappings. This isn’t desirable, as you’d be adding complexity simply to satisfy the ORM. Sometimes that may be necessary, but when it’s not I prefer to avoid it.

It would be simpler to just let go of any dependency on the EF relationship magic and control the setting of the foreign key in your code.

The first step is to tell EF to ignore this relationship; otherwise, it will continue to look for a foreign key.

Here’s code I’ll use in the DbContext.OnModelBuilder method override so that EF won’t pay attention to that relationship:

modelBuilder.Entity<Order>().Ignore(o => o.LineItems);

Now, I’ll take control of the relationship myself. This means refactoring so I add a constructor to LineItem that requires OrderId and other values, and it makes LineItem much more like a DDD entity so I’m happy. I also have to modify the CreateLineItem method in Order to use that constructor rather than an object initializer.

**Figure 2** shows an updated version of the repository method.

**Figure 2 The Repository Method**

public void UpdateLineItemsForExistingOrder(Order order)

{

  foreach (var item in order.LineItems)

  {

    if (item.LineItemId > 0)

    {

      \_context.Entry(item).State = EntityState.Modified;

    }

    else

    {

      \_context.Entry(item).State = EntityState.Added;

      item.SetOrderIdentity(order.OrderId);

    }

  }

}

Notice I’m no longer adding the order graph and then fixing the order’s state to Unchanged. In fact, because EF is unaware of the relationship, if I called context.Orders.Add(order), it would add the order instance but wouldn’t add the related line items as it did before.

Instead, I’m iterating through the graph’s line items and not only setting the state of existing line items to Modified but setting the state of new ones to Added. The DbContext.Entry syntax I’m using does two things. Before it sets the state, it checks to see if the context is already aware of (or “tracking”) that particular entity. If it’s not, then internally it attaches the entity. Now it’s able to respond to the fact that the code is setting the state property. So in that single line of code, I’m attaching and setting the state of the LineItem.

My code is now in accord with another healthy prescription for using EF with DDD, which is: don’t rely on EF to manage relationships. EF performs a lot of magic, a huge bonus in many scenarios. I’ve happily benefited from this for years. But for DDD aggregates, you really want to manage those relationships in your model and not rely on the data layer to perform necessary actions for you.

Because I’m stuck for the time being using integers for my keys (Order.OrderId, for example) and depending on my database to provide the values of those keys, I need to do some extra work in the repository for new aggregates such as a new order with line items. I’ll need tight control of the persistence so I can use the old-fashioned pattern of inserting graphs: insert order, get new database-generated OrderId value, apply that to the new line items, and save them to the database. This is necessary because I’ve broken the relationship that EF would normally use to perform this magic. You can see in the sample download how I’ve implemented this in the repository.

I’m ready, after many years, to stop depending on the database to create my identifier and begin to use GUIDs for my key values, which I can generate and assign in my app. This allows me to further separate my domain from the database.

## Keeping the EF Relationship-Management Magic—for Queries

Divesting my model of EF relationships really helped in the previous scenario for performing updates. But I don’t want to lose all of the relationship features of EF. Loading related data when querying from the database is one feature I don’t want to give up. Whether I’m eager loading, lazy loading or explicitly loading, I love benefiting from the ability of EF to bring related data along without having to express and execute additional queries.

This is where an extended view of the separation of concerns concept comes into play. When following DDD precepts for design, it’s not unusual to have different representations of similar classes. For example, you might do this with a Customer class designed to be used in the context of customer management, as opposed to a Customer class for simply populating a pick list that needs only the customer’s name and identifier.

It also makes sense to have different DbContext definitions. In scenarios where you’re retrieving data, you might want a context that’s aware of the relationship between Order and LineItems so you can eagerly load an order along with its line items from the database. But then, when you’re performing updates as I did earlier, you may want a context that explicitly ignores that relationship so you can have more granular control of your domain.

An extreme view of this for a certain subset of complex problems you may be solving with software is a pattern called Command Query Responsibility Segregation (CQRS). CQRS guides you to think of data retrieval (reads) and data storage (writes) as separate systems that may require distinct models and architectures. My small example, which highlights the benefit of having the data-retrieval operations embrace a different understanding of relationships than data-storage operations, gives you an idea of what CQRS can help you achieve. You can learn more about CQRS from the excellent resource, CQRS Journey, available at [msdn.microsoft.com/library/jj554200](https://msdn.microsoft.com/library/jj554200).

## Data Access Happens in the Repository, Not the Aggregate Root

I want to back up a bit now and tackle one last question that gnawed at me when I started focusing on unidirectional relationships. (This is not to say that I have no more questions about DDD, but this is the final topic I’ll address in this series.) This question about unidirectional relationships is a common one for us “database-first” thinkers: Where, exactly (with DDD), does data access take place?

When EF was first released, the only way it could work with a database was to reverse-engineer an existing database. So, as I noted earlier, I got used to every relationship being two-way. If the Customers and Orders tables in the database had a primary key/foreign key constraint describing a one-to-many relationship, I saw that one-to-many relationship in the model. Customer had a navigation property to a collection of orders. Order had a navigation property to an instance of Customer.

As things evolved to Model- and Code-First, where you can describe the model and generate a database, I continued to follow that pattern, defining navigation properties on both ends of a relationship. EF was happy, mappings were simpler and coding was more natural.

So, with DDD, when I found myself with an Order aggregate root that was aware of CustomerId or maybe even a full Customer type, but I couldn’t navigate from Order back to Customer, I got upset. The first question I asked was, “what if I want to find all of the orders for a customer?” I always assumed I’d need to be able to do that, and I was used to relying on having access to navigation in both directions.

If logic begins with my order aggregate root, how would I ever answer that question? I also initially had the misconception that you do everything through the aggregate root, which didn’t help.

The solution made me hit my head and feel a bit foolish. I share my foolishness here in case someone else gets stuck in the same way. It’s not the job of the aggregate root, nor the job of the Order, to help me answer that question. However, in an Order-focused repository, which is what I’d use to perform my queries and persistence, there’s no reason I can’t have a method to answer my question:

public List<Order>GetOrdersForCustomer(Customer customer)

  {

    return \_context.Orders.

      Where(o => o.CustomerId == customer.Id)

      .ToList();

  }

The method returns a list of Order aggregate roots. Of course, if I’m creating this in the scope of doing DDD, I’d only bother putting that method in my repository if I know it’s going to be needed in the particular context, not “just in case.” Chances are, I’d need it in a reporting app or something similar, but not necessarily in a context designed for building sales orders.

## Only the Beginning of My Quest

As I’ve learned about DDD over the past few years, the topics I covered in this series are the ones that I had the most difficulty either comprehending or figuring out how to implement when Entity Framework would be part of my data layer. Some of the frustration I encountered was due to years of thinking about my software from the perspective of how things would work in my database. Letting go of this perspective has been freeing because it lets me focus on the problem at hand—the domain problem for which I’m designing software. At the same time, I do need to find a healthy balance because there may be data-layer issues I encounter when it’s time to add that into my solution.

While I’ve focused on how things might work when I’m mapping my classes directly back to the database with Entity Framework, it’s important to consider that there could be another layer (or more) between the domain logic and the database. For example, you might have a service with which your domain logic interacts. At that point, the data layer is of little (or no) consequence to mapping from your domain logic; that problem now belongs to the service.

There are many ways to approach your software solutions. Even when I’m not implementing a full end-to-end DDD approach (something that takes quite a bit of mastery), my entire process continues to benefit from the lessons and techniques I’m learning from DDD.