Part 2

title

text

8 Domain model

This chapter covers

- Designing domain models
- Exploring a real-world domain model
- Understanding entities and value objects
- Thinking about persistence

In chapter 2 we explored the M in MVC—the presentation model our controllers beam through a prism of markup, refracted onto the screen by the view. For the most part, the presentation model doesn't contain any behavior. Its power is in its shape and structure, not in its algorithms and interactions. The presentation model serves the user interface.

Deeper, toward the application's core, there's another focus: the logic and code that do the work. The core also contains the valuable calculations and business rules that make the application worth using. In an ecommerce application, this focus might be on orders and products, and in a hotel management system the focus might be on reservations and rooms. This other focus—we'll call it the application's *domain*—deserves a model too: the domain model.

In this chapter, we'll explore a sample model for a simple system that manages a small ecommerce business. The model enables the application to provide an interesting service. Without the model, the application provides no value. We place great importance on creating a rich model that clearly expresses the business reality and the solution to problems in that domain.

The style of modeling we'll use in this book is *domain-driven design* (DDD), as conveyed by Eric Evans in his book, *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Covering the topic in depth takes a book in itself; we'll tackle a small primer, which should enable you to follow the software examples in the rest of this book. After the DDD primer, we'll discuss how to best use the domain model, and then we'll look at how to use a presentation model to keep controllers and views simple. We'll keep a keen eye on separation of concerns to ensure that every class has a single, well-defined responsibility.

8.1 Understanding the basics of domain-driven design

Developers can use different methods to model software. The method we prefer is domain-driven design (DDD), which looks at the business domain targeted by the software and models objects to represent the various entities and the relationships between the entities.

We refer to the domain model as the *object graph* that represents the business domain of the software. If the software lives in the online ecommerce space, we'd expect to find objects such as `Order`, `Customer`, `Product`, and so on. These aren't just *data-transfer objects*; they're rich objects with properties and methods that mimic behavior in that business space. Popular in .NET development, the `DataSet` object wouldn't be appropriate in a domain model because the `DataSet` is a relational representation of database tables. Whereas the `DataSet` is a model focused on data relationships and persistence, a domain model is focused more on behavior and responsibility. In our fictitious ecommerce domain, shown in chapter 2 (figure 2.1), when retrieving order history for a customer, we want to retrieve an array or collection of `Order` objects, not a `DataSet` of order data. The heavy focus on the separation of behavior and the encapsulated view of data is key in DDD.

A note about routing

If you're unfamiliar with DDD, you may want to review some of the following references. Reviewing these publications isn't necessary for the purposes of this book, but they'll help you as you develop software in your career. From this point forward, we'll defer to these resources for more detail on domain models, bounded contexts, aggregates, aggregate roots, repositories, entities, and value objects. When discussing each of these concepts, we'll talk only briefly about their purpose and then move on.

- *Domain-Driven Design: Tackling Complexity in the Heart of Software* by Eric Evans. The most complete reference for DDD. Evans can be credited with making this collection of patterns known. He applies his own experience as he names patterns that work together to simplify complex software. (Addison-Wesley Professional, 2003.)

(continued)

- *Domain Driven Design Quickly* by Abel Avram Floyd Marinescu. A 104-page book designed to be a more concise guide to DDD than Evans' book. This ebook is summarized mainly from Evans' book. (Lulu Press, 2007.)
- *Applying Domain-Driven Design and Patterns: With Examples in C# and .NET* by Jimmy Nilsson. The author takes the reader through real, complete examples and applies DDD patterns along with test-driven development (TDD) and O/R mapping. (Addison-Wesley Professional, 2006.)
- Domain-Driven Design Community (http://domaindrivendesign.org/). An evolving information website maintained by Eric Evans, Jimmy Nilsson, and Ying Hu.

8.2 A sample domain model

We included a sample domain model in the example code for this book. In figure 8.1, you can see this sample domain model, and we'll work with different pieces of it in the rest of this chapter.

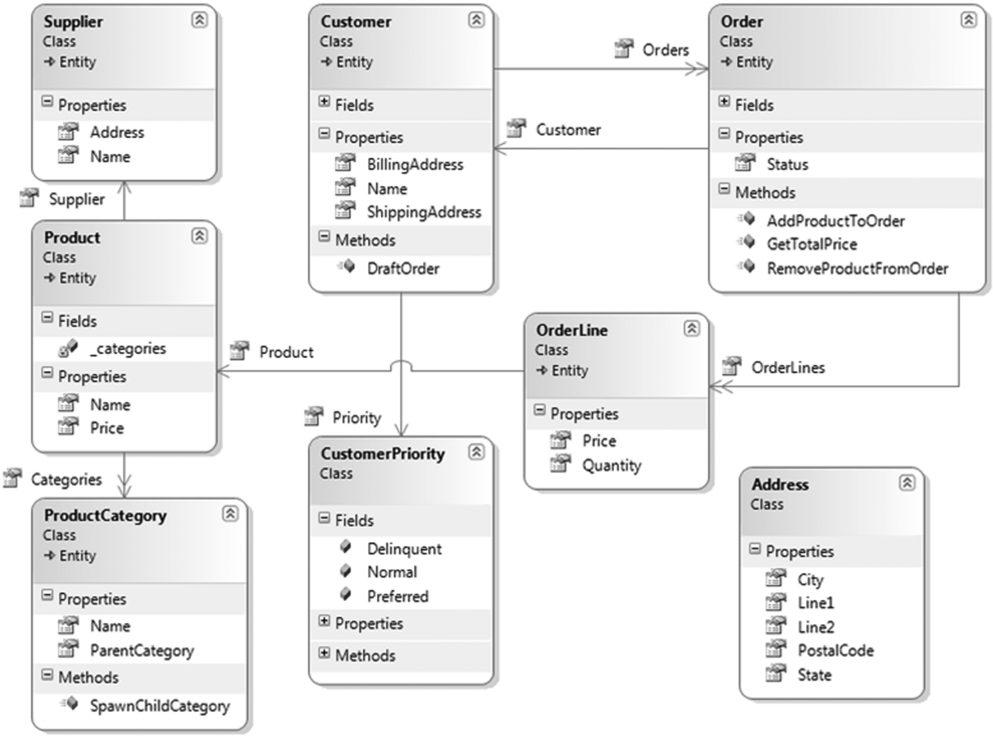

Figure 8.1 An example domain model

8.2.1 Key entities and value objects

Figure 8.1 shows some of the entities and value objects in play within our domain model. The entities are the important objects in our domain model, such as `Customer`, `Order`, `Product`, and `Supplier`. With so many types in the diagram, you're probably wondering what's special about these classes and what makes them entities.

The defining characteristic of an entity is that it has the concept of an identity, a property that can be examined to determine uniqueness. The reason we give these objects an identifier is that they can stand on their own, and we can speak about these objects without other supporting concepts. It would make sense to list a collection of any of these objects. Entities can stand on their own, and we can think about them in a collection or as a single object.

Value objects don't make sense on their own without the supporting context of an entity to which they belong. Some value objects in our domain model are `CustomerPriority` and `Address`. Also, many properties of entities are value objects. Let's discuss `CustomerPriority` and what context is required for it to make any sense.

A `CustomerPriority` has a value that indicates the priority level of the customer. It belongs completely to the `Customer` class; without `Customer`, `CustomerPriority` would have no context and would have no meaning. As a value object, `CustomerPriority` is defined by its properties and methods and has no identifier. It wouldn't make sense to list a collection or array of `CustomerPriority` instances because, without the `Customer`, it has no meaning or purpose. Its relationship with other entities gives it meaning. The `Customer` it belongs to and the status information it includes give it the context to convey meaning in the application, and when some other code needs the customer's priority, it must ask the `Customer` instance for the `CustomerPriority`. The `Customer` object will hand back this object.

Like `CustomerPriority`, other types without identifiers are value objects. Value objects aren't glamorous, and even describing them can be boring. The arrangement of entities and value objects into larger structures can be interesting.

Entities and value objects are useful in separating responsibilities in a domain model, but there's more. If we need to load a `Product` entity, what does that mean? We see that our `Product` object can have many `ProductCategory(s)`, and that each `ProductCategory` has a `parent ProductCategory`. Going further, a `Product` has a `Price` property. `Orders and Suppliers` all have a relationship with a `Product`. When we need to deal with a `Product` object, must we have all associated objects in memory for any operation to make sense? The answer is no. In DDD, we divide our domain model into what are called *aggregates.*

8.2.2 Aggregates

Aggregates are groups of objects that work and live together. We group them along natural operational lines, and one entity serves as the *aggregate root.* The aggregate root is the entry point and the hub of operations for all objects in the aggregate. An aggregate can have many objects, or it can just be a single entity, but the aggregate root is always an entity because it must be able to stand on its own, and only entities can stand on their own. In figure 8.2, we see the `Order` aggregate.

The aggregate root is the `Order` class, and another member of the `Order` aggregate is `OrderLine`. This isn't the complete `Order` aggregate, but it demonstrates some conventions of the aggregate pattern. It may seem trivial that we classify this object in the `Order` aggregate, but specifying ownership is valuable. We've specified that the `Order` type owns the types in the `Order` aggregate. Objects in other aggregates aren't allowed to have a durable (non-transient) reference with the non-root objects in the `Order` aggregate.

NOTE `OrderLine` holds a reference to `Product`, which is another aggregate root. Types in an aggregate are allowed to hold references to other aggregate roots only, not to other non-root types in a different aggregate. For instance, a `Supplier` wouldn't hold a reference to an `OrderLine` because `OrderLine` is a non-root type in the `Order` aggregate. In short, if a type belongs to an aggregate, types in other aggregates must not hold a durable reference.

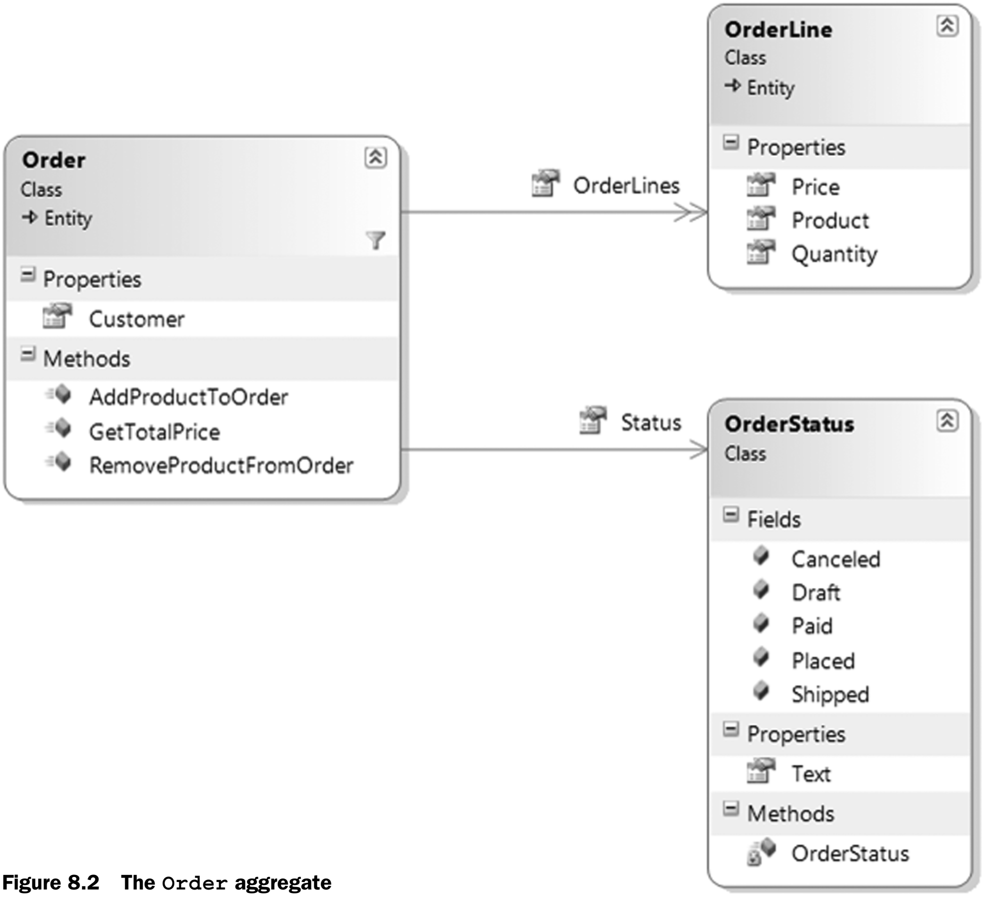

Figure 8.2 The `Order` aggregate

The separation into aggregates enables the application to work with domain objects easily. If we didn't draw aggregate boundaries, the entire domain model could easily devolve into a ball of spaghetti references. Conceivably, we wouldn't be able to use any objects without the entire object graph loaded into memory.

Aggregate boundaries help us define how much of the domain model is necessary for an interesting operation. For instance, if we want to build a presentation model with `Customer` information by `Order`, we don't need to load the entire object graph. We only need the `Order` aggregate and the other aggregate roots that are necessary. In fact, if we need only the status of the order, we wouldn't even have to load the entire `Order` aggregate.

Now that we're discussing how much of the object graph to load, you might wonder why we haven't yet discussed persistence to a database.

8.2.3 Persistence for the domain model

For this book, persistence is just not that interesting. ASP.NET MVC is a UI framework, so it can be used with or without a database. Sure, we can imagine how we might load and save these objects from and to a relational database, XML files, web services, and so on, but when designing a domain model, persistence concerns are mostly orthogonal to the model. For most business applications, we'll have to durably save the state of the application somehow, but the domain model shouldn't have to care whether that persistence is to XML files, a relational database, an object database, or to memory.

NOTE Persistence is interesting and necessary for real applications. We aren't discussing specific data access techniques because that topic is orthogonal to the ASP.NET MVC Framework. The MVC Framework is a presentation-layer concern, and it can work with many data access strategies. Your backend data access decisions don't change if you use the ASP.NET MVC Framework instead of Web Forms, Windows Forms, WPF, Silverlight, or even a console UI. If this is of immediate interest, take a peek ahead at chapter 23, which shows how to use NHibernate with an ASP.NET MVC UI.

Regardless of the persistence mechanism, the domain model includes a concept for loading and saving object state. Notice that we're not talking about loading and saving data. In the domain model, we're concerned with objects, not data. We need to load object state and persist object state, and we do that using *repository* types. In DDD, we dedicate a repository to each aggregate, and the repository is responsible for loading and saving object state. The repository performs the operations on the aggregate root only.

In the case of the `Order` aggregate, we'll work with a type called `IOrderRepository`. In figure 8.3, we see the repository whose responsibility it is to perform persistence operations on the `Product` aggregate.

Let's examine the `Order` aggregate once again as it relates to persistence. Suppose that when using this application we add several items, `OrderLines`, to our cart. In the application, we'd add `OrderLine` instances to our `Order` instance and then pass our `Order` to the `Save()` method of `IOrderRepository`. The repository would

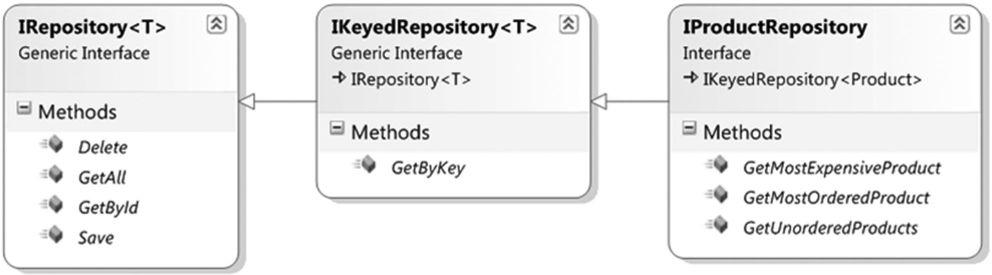

Figure 8.3 IProductRepository—all persistence operations on the aggregate root

be responsible for saving the OrderLine instances as well, because these objects live within the Order aggregate. The repository's responsibility is to manage persistence for the Order aggregate, which means every object in the aggregate.

The repository interfaces will provide the objects we need to work with for all the examples in this book, and the controller classes will depend on these repository interfaces as well as other logical service types. Because data access and screen controllers have completely different responsibilities, a screen controller in this book will never concern itself with how any sort of data access is performed, or with whether data access is happening at all. A screen controller will call methods on dependencies, which will often be repositories, and when calling the Save() method on IOrderRepository, for example, the screen controller doesn't care whether the implementation saves the object in an in-memory cache, an XML file, or a relational database. The controller will merely call the repository and trust that what's behind the interface will work appropriately.

NOTE No doubt you have seen some examples where controller actions directly contain data access code. With LINQ to SQL easy to use and growing in popularity, conference talks are featuring ASP.NET MVC Framework demos where a controller action performs a LINQ to SQL query. This works for small or short-lived applications, but it's inappropriate for long-lived business applications because of the coupling. For years, the industry has known that coupling presentation with data access is a recipe for disaster. These concepts gave birth to the well-known *data access layer*. When using the ASP.NET MVC Framework, a controller is part of the presentation layer. The best practice is still to avoid putting data access in your presentation layer, any data access concern in a controller action creates technical debt that will put a tax on maintenance for the life of the application.

One benefit that we can capitalize on immediately when separating our data access layer from the presentation and business layers is unit testing. While unit testing our screen controllers, you'll notice we frequently fake out the repository interfaces so that they return a canned list of objects as the context for a test. Unit-testing controllers should never involve any persistence mechanism or exercise external dependencies. We covered unit testing of controllers in detail in chapter 4, but in a unit test, the repository implementation will never come into play. A test double, or substitute object, will always be provided for the interface.

8.3 Summary

In this chapter, we learned about a richer, more functional model we use to represent the real-world problems and things our application manages. We learned about the different types of domain objects and how we can group those objects into aggregates to specify logical boundaries. We learned about abstracting persistence with repositories, where queries are expressed as methods in the domain language.

In the next chapter, we'll tread deep into controller territory, exploring ASP.NET MVC 2 features and extensibility points that will be our technical base for success with the framework.