8

Domain Model

This chapter covers

* Guidance for designing the model
* Exploring a real-world domain model
* Entities and value objects
* Thinking about persistence

Without a model, software is not interesting. A “model” in the English language is just like a model in software: a representation of the real thing. In software, we represent the real world by using objects that are named after concepts we deal with every day. These objects have attributes and behaviors similar to those found in the real world. In this chapter, we’ll explore a sample model for a typical system that might manage a small ecommerce application. 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 with which our controllers can work.

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 is 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, then we’ll move through how to use a presentation model to keep controllers and views simple. We’ll keep a keen eye on SoC, and we’ll ensure that every class has a single, well-defined responsibility. Before digging deep, we need a good understanding of the basics of DDD.

2.1 Understanding the basics of domain-driven design

Developers can use different methods to model software. The method we prefer is domain-driven design which looks at the business domain targeted by the software and models objects to represent the appropriate concepts. 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 would expect to find objects such as Order, Customer, Product, etc. These are not just data-transfer objects either. They are rich objects with properties and methods that mimic behavior in that business space. Popular in .NET development, the DataSet object would not be appropriate in a domain model because the DataSet is a relational representation of database tables. Whereas the DataSet is a model focused on the data relationships and persistence, a domain model is focused more on behavior and responsibility.

In our fictitious ecommerce application, when retrieving order history for a customer, we would want to retrieve an array or collection of Order objects, not a DataSet of order data. The heavy focus on the demarcation of behavior and the encapsulated view of data is key in DDD. If you are unfamiliar with domain-driven design, you may want to review some of the following references. Reviewing these publications is not necessary for the purpose of this book, but they will 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, 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. The next section is an overview of the core domain model for this book.

References for learning more

Domain-Driven Design: Tackling Complexity in the Heart of Software by Eric Evans. The most complete reference for DDD. Evans can be credited for 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).

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, Inc. (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).

<http://domaindrivendesign.org/>, an evolving, information website maintained by Eric Evans, Jimmy Nilsson, and Ying Hu.

2.2 A sample domain model

Throughout the rest of this book, our examples will be centered on the open source project, CodeCampServer. Authors of this book started the project, and it is being extended at the time of publishing by a strong network of contributors. This software can serve as the official website for a software conference, often called a Code Camp. The domain model is centered on the concepts present when managing a Code Camp. Since “Code Camp” is a common name (also common is “TechFest” or “BarCamp”), our central object is Conference. In figure 2.1, you see the complete domain model for the application, and we’ll work with different pieces in the examples following in the chapter and the rest of the book.

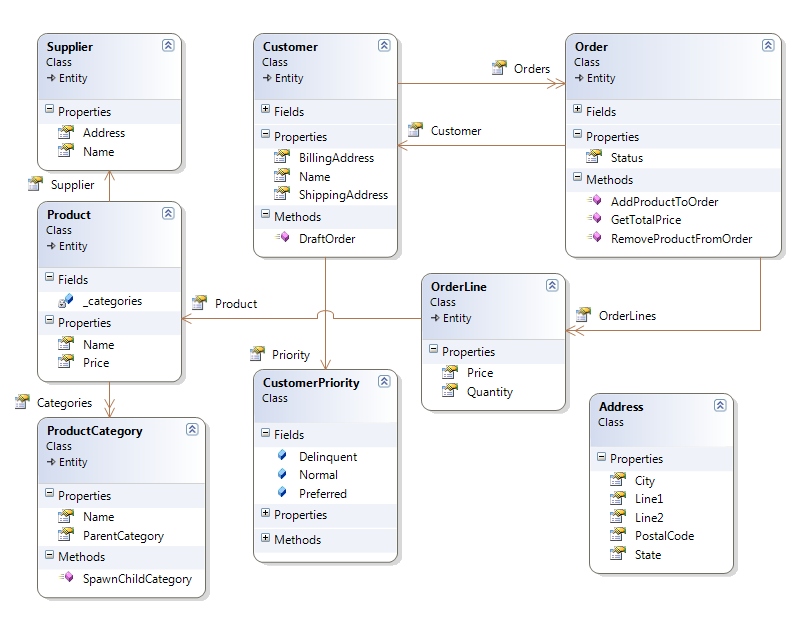


Figure 2.1 Partial domain model for CodeCampServer, which will serve as the basis of future examples in this book.

2.2.1 Key entities and value objects

Figure 2.1 shows some of the entities and value objects in play within our domain model. The entities are the key objects in our domain model such as: Conference, Session, Track, and TimeSlot. With so many types in the diagram, you probably wonder what is special about these classes and what makes them entities. The reason these are entities is that they have the concept of an identity, a property which can be examined to determine uniqueness. The reason we give these objects an identifier is that these 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 reason 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 Session Level, and Conference Address. Also many properties of entities are value objects. Let’s discuss Level and what context is required for it to make any sense.

A Level has a value that indicates the difficulty level of the session. It does not have an identifier. Level belongs completely to the Session class. Without Session, Level would have no context and would have no meaning. The purpose of Level is to denote the information that helps attendees of the conference choose what sessions may be appropriate. Being a value object, Level is defined by its properties and methods and has no identifier. It would not make sense to list out a collection or array of Level instances because without the Session, it has no meaning or purpose. Its relationship with other entities gives it meaning. The Session it belongs to and the difficulty level information it includes give it the context to convey meaning in the application, and when some other code needs the session’s Level, it must ask the Session instance for the Level. The Session object will hand back this object. Like Level, other types without identifiers are value objects. Value objects are not 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 is more. If we need to load a Conference entity for the Austin .NET User Group Code Camp, what does that mean? We see that our Conference object can have many Sessions, and that each Session has a Speaker. Going further, a Speaker has a WebsiteUrl property. Tracks, Sessions, and Attendees all have a relationship with a Conference. When we need to deal with a Conference 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 aggregates.

2.2.2 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 serves as 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 since it must be able to stand on its own, and only entities can stand on their own. In figure 2.2, we see some of the aggregates for CodeCampServer, with the Conference aggregate in the center.

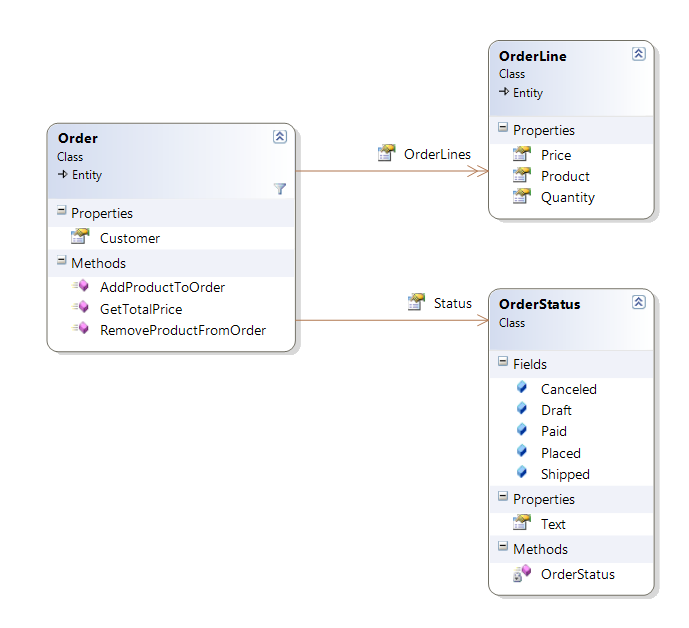


Figure 2.2 The Conference aggregate

The aggregate root is the Conference class, and another member of the Conference aggregate is Attendee. This is not the complete Conference aggregate, but it demonstrates some conventions of the aggregate pattern. It may seem trivial that we classify this object in the Conference aggregate, but specifying ownership is valuable. We have specified that the Conference type owns the types in the Conference aggregate. Objects in other aggregates are not allowed to have a durable relationship with the nonroot objects in the Conference aggregate.

Note

Session holds a reference to Track, which is another aggregate root. Types in an aggregate are allowed to hold references to other aggregate roots only, not to other nonroot types in a different aggregate. For instance, even if only five Attendees could attend a session, Session would not be allowed to have a reference to the several Attendee instances because Attendee is a nonroot type in the Conference aggregate. In short, if a type belongs to an aggregate, types in other aggregates must not hold a durable reference.

The separation into aggregates enables the application to work with domain objects easily. If we did not 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 to define how much of the domain model is necessary for an interesting operation. For instance, if we want to show conference information on a screen with the location, directions, sessions, and speakers, we don’t need to load the entire object graph. We only need the Conference aggregate and the other aggregate roots that are necessary. In fact, if we need only the start and end date for the conference, we would not even have to load the entire Conference aggregate. Now that we are discussing how much of the object graph to load, you might wonder why we haven’t yet discussed persistence to a database.

2.2.3 Persistence for the domain model

For this book, persistence is just not that interesting. 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 should not have to care whether that persistence is to XML files, a relational database, an object database, or if the entire state of the application is just kept around in memory.

note

Persistence is interesting and necessary for real applications. We are not 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 back-end data access decisions do not change if you use the ASP.NET MVC Framework instead of Web Forms, Windows Forms, WPF, Silverlight, or even a console UI.

Regardless of the persistence mechanism, the domain model includes a concept for loading and saving object state. Notice how we are not talking about loading and saving data. In the domain model, we are concerned about objects, not data. We need to load object state and persist object state. We do that using repository types. In domain-driven design, 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 Conference aggregate, we’ll work with a type called IConferenceRepository. In figure 2.3, we see the repository whose responsibility it is to perform persistence operations on the Conference aggregate.

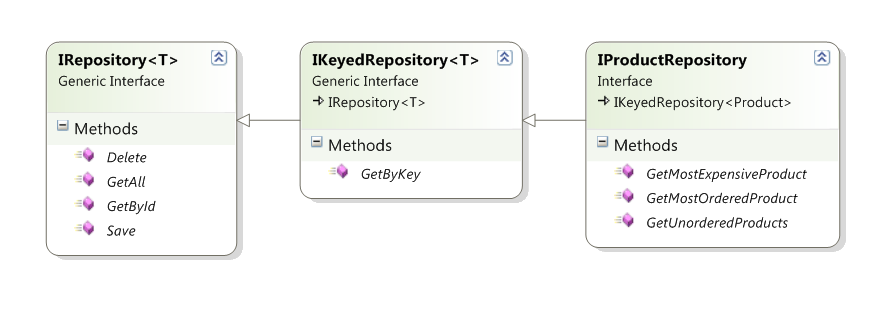


Figure 2.3 IConferenceRepository–all persistence operations on the aggregate root

For more examples, we have a repository for each aggregate in our domain model. Some of them are:

* IConferenceRepository–Persistence operations on the Conference aggregate
* ISessionRepository–Persistence operations on the Session aggregate
* ITimeSlotRepository–Persistence operations on the TimeSlot aggregate
* ITrackRepository–Persistence operations on the Session Track aggregate

Let’s examine the Conference aggregate once again as it relates to persistence. Suppose that when managing a Code Camp with this application we add several attendees. In the application we would add Attendee instances to our Conference instance and then pass our Conference to the Save() method of IConferenceRepository. The repository would be responsible for saving the Attendee instances as well because these objects live within the Conference aggregate. The repository’s responsibility is to manage persistence for the Conference aggregate, which means every object in the aggregate.

You are probably wondering what mechanism we are using for persistence because we still have not mentioned it. With this book, you can download the full source code and examine the classes that implement our repository interfaces, but for the purpose of exploring the ASP.NET MVC Framework, we find it irrelevant and a distraction to explore the data access code, and we’ll keep this book’s focus on the presentation layer, which is where the ASP.NET MVC Framework lives. 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. Since data access and a screen controller have completely different concerns, a screen controller in this book will never concern itself with how any sort of data access is performed, or that 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 IConferenceRepository, the screen controller does not 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 is 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 being new 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 is inappropriate for long-lived business applications because of the coupling. For years, the industry has known that coupling presentation concerns with data access concerns 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 still stands 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 will 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’ll cover the unit testing of controllers in much more detail in chapter 3, but in a unit test, the repository implementation will never come into play. A substitute object will always be provided for the interface.

At this point, we have enough information about our domain model to proceed, but the domain model is not the only type of model that we need. The domain model is important because it represents unique concepts in the real world as they really are. A conference can have many attendees, so that is how we model it. An attendee describes a person who is coming to the conference, and that is how we represent it in code. Now, what about a schedule listing? When listing the time slots, sessions, and speakers, how do we work with that in the presentation layer (in our screens)?