# Chapter 13: Designing Business Entities

01/13/2010 • 11 minutes to read

For more details of the topics covered in this guide, see Contents of the Guide.

# Contents



# Overview

Business entities store data values and expose them through properties; they contain and manage business data used by an application and provide stateful programmatic access to the business data and related functionality. Business entities should also validate the data contained within the entity and encapsulate business logic to ensure consistency and to implement business rules and behavior. Therefore, designing or choosing appropriate business entities is vitally important for maximizing the performance and efficiency of your business layer.

This chapter will help you to understand the design of business entity components. It starts by looking at different data formats and how data will be used in your application. Next, you will learn how the data format you choose determines the way that business rules can be implemented in your design. Finally, you will learn about design options for custom objects, and how to support serialization with different data formats.

For general component design considerations, and more information on the components commonly found in the layers of an application, see Chapter 10 "Component Guidelines."

# Step 1 – Choose the Representation

In this step, you will learn about the different ways of representing business entities, and see the benefits and liabilities of each, to help you choose the correct representation for your scenario. The following list describes the most common format options:

- **Custom Business Objects**. These are common language runtime (CLR) objects that describe entities in your system. An object/relational mapping (O/RM) technology such as the ADO.NET Entity Framework (EF) or NHibernate can be used to create these objects (for more information, see Additional Resources at the end of this chapter). Alternatively, you can create them manually. Custom business objects are appropriate if you must encapsulate complex business rules or behavior along with the related data. If you need to access your custom business objects across **AppDomain**, process, or physical boundaries, you can implement a service layer that provides access via Data Transfer Objects (DTO) and operations that update or edit your custom business objects.
- **DataSet** or **DataTable**. **DataSets** are a form of in-memory database that usually maps closely to an actual database schema. **DataSets** are typically used only if you are not using an O/RM mapping mechanism and you are building a data-oriented application where the data in your application logic maps very closely to the database schema. **DataSets** cannot be extended to encapsulate business logic or business rules. Although **DataSets** can be serialized to XML, they should not be exposed across process or service boundaries.
- **XML**. This is a standards-based format that contains structured data. XML is typically used to represent business entities only if your presentation layer requires it or if your logic must work with the content based on its schema; for example, a message routing system where the logic routes messages based on some well-known nodes in the XML document. Be aware that using and manipulating XML can use large amounts of memory.

# Step 2 – Choose a Design for Business Entities

If you have determined that custom objects provide the best representation for business entities, the next step is to design those objects. The design approach used for custom objects is based on the sort of object that you plan to use. For example, domain model entities require in-depth analysis of a business domain, while table module entities require an understanding of the database schema. The following is a list of common design approaches when using business objects:

- **Domain Model** is an object-oriented design pattern. The goal in a domain model design is to define business objects that represent real world entities within the business domain. When using a domain model design, the business or domain entities contain both behavior and structure. In other words, business rules and relationships are encapsulated within the domain model. The domain model design requires in-depth analysis of the business domain and typically does not map to the relational models used by most databases. Consider using the domain model design if you have complex business rules that relate to the business domain, you are designing a rich client and the domain model can be initialized and held in memory, or you are not working with a stateless business layer that requires initialization of the domain model with every request. For more information on the Domain Model and Domain-Driven Design, see the section "Domain-Driven Design" later in this chapter.
- **Table Module** is an object-oriented design pattern. The objective of a table module design is to define entities based on tables or views within a database. Operations used to access the database and populate the table module entities are usually encapsulated within the entity. However, you can also use data access components to perform database operations and populate table module entities. Consider using the table module approach if the tables or views within the database closely represent the business entities used by your application, or if your business logic and operations relate to a single table or view.
- **Custom XML objects** represent deserialized XML data that can be manipulated within your application code. The objects are instantiated from classes defined with attributes that map properties within the class to elements and attributes within the XML structure. The Microsoft .NET Framework provides components that can be used to deserialize XML data into objects and serialize objects into XML. Consider using custom XML objects if the data you are consuming is already in XML format (for example, XML files or database operations that return XML as the result set); you need to generate XML data from non-XML data sources; or you are working with read-only document-based data.

When using custom objects, your business entities are not all required to follow the same design. For example, one aspect of the application with complex rules may require a Domain Model design. However, the remainder of the application may use XML objects, a Table Module design, or domain objects as appropriate.

# Step 3 – Determine Serialization Support

You must determine how you will transfer business entities across boundaries. In most cases, to pass data across physical boundaries such as **AppDomain**, process, and service interface boundaries, you must serialize the data. You may also decide to serialize the data when crossing logical boundaries; however, keep in mind the performance impact in this case. Consider the following options for transferring business entities:

- **Expose serializable business entities directly only if required**. If another layer in your application, on the same physical tier, is consuming your business entities, the most straightforward approach is to expose your business entities directly through serialization. However, the disadvantage of this approach is that you create a dependency between the consumers of your business entities and their implementation. Therefore, this approach is not generally recommended unless you can maintain direct control over the consumers of your business entities and remote access to your business entities between physical tiers is not required.
- **Convert business entities into serializable data transfer objects**. To decouple the consumers of your data from the internal implementation of your business layer, consider translating business entities into special serializable data transfer objects. Data Transfer Object (DTO) is a design pattern used to package multiple data structures into a single structure for transfer across boundaries. Data transfer objects are also useful when the consumers of your business entities have a different data representation or model, for example a presentation tier. This approach makes it possible to change the internal implementation of the business layer without affecting any consumers of the data, and allows you to version your interfaces more easily. This approach is recommended when having external clients consuming data.
- **Expose XML directly**. In some cases, you may serialize and expose your business entities as XML. The .NET Framework provides extensive serialization support for XML data. In most cases, attributes on your business entities are used to control serialization into XML.

For more information about data schemas for service interfaces, see Chapter 9 "Service Layer Guidelines." For more information about communicating between layers and tiers, see Chapter 18 "Communication and Messaging."

# Domain Driven Design

Domain Driven Design (DDD) is an object-oriented approach to designing software based on the business domain, its elements and behaviors, and the relationships between them. It aims to enable software systems that are a realization of an underlying business domain by defining a domain model expressed in the language of business domain experts. The domain model can be viewed as a framework from which solutions can then be rationalized.

To apply Domain Driven Design, you must have a good understanding of the business domain you want to model, or be skilled in acquiring such business knowledge. The development team will often work with business domain experts to model the domain. Architects, developers, and subject matter experts have diverse backgrounds, and in many environments will use different languages to describe their goals, designs and requirements. However, within Domain Driven Design, the whole team agrees to only use a single language that is focused on the business domain, and which excludes any technical jargon.

As the core of the software is the domain model, which is a direct projection of this shared language, it allows the team to quickly find gaps in the software by analyzing the language around it. The creation of a common language is not merely an exercise in accepting information from the domain experts and applying it. Quite often, communication problems within development teams are due not only to misunderstanding the language of the domain, but also due to the fact that the domain's language is itself ambiguous. The Domain Driven Design process holds the goal not only of implementing the language being used, but also improving and refining the language of the domain. This in turn benefits the software being built, since the model is a direct projection of the domain language.

The domain model is expressed using entities, value objects, aggregate roots, repositories, and domain services; organized into coarse areas of responsibility known as Bounded Contexts.

**Entities** are objects in the domain model that have a unique identity that does not change throughout the state changes of the software. Entities encapsulate both state and behavior. An example of entity could be a Customer object that represents and maintains state about a specific customer, and implements operations that can be carried out on that customer.

**Value objects** are objects in the domain model that are used to describe certain aspects of a domain. They do not have a unique identity and are immutable. An example of value object could be a Transaction Amount or a Customer Address.

**Aggregate Roots** are entities that group logically related child entities or value objects together, control access to them, and coordinate interactions between them.

**Repositories** are responsible for retrieving and storing aggregate roots, typically using an Object/Relational Mapping (O/RM) framework.

**Domain services** represent operations, actions, or business processes; and provide functionality that refers to other objects in the domain model. At times, certain functionality or an aspect of the domain cannot be mapped to any objects with a specific lifecycle or identity; such functionality can be declared as a domain service. An example of a domain service could be catalog pricing service within the e-commerce domain.

In order to help maintain the model as a pure and helpful language construct, you must typically implement a great deal of isolation and encapsulation within the domain model. Consequently, a system based on Domain Driven Design can come at a relatively high cost. While Domain Driven Design provides many technical benefits, such as maintainability, it should be applied only to complex domains where the model and the linguistic processes provide clear benefits in the communication of complex information, and in the formulation of a common understanding of the domain.

For a summary of domain driven design techniques, see "*Domain Driven Design Quickly*" at http://www.infoq.com/minibooks/domain-driven-design-quickly. Alternatively, see "*Domain-Driven Design: Tackling Complexity in the Heart of Software*" by Eric Evans (Addison-Wesley, ISBN: 0-321-12521-5) and "*Applying Domain-Driven Design and Patterns*" by Jimmy Nilsson (Addison-Wesley, ISBN: 0-321-26820-2).

# Additional Resources

For more information, see the following resources:

- For more information on design patterns for business entities, see "*Enterprise Solution Patterns Using Microsoft .NET*" at http://msdn.microsoft.com/en-us/library/ms998469.aspx.
- For more information on designing business entities, see "*Integration Patterns*" at http://msdn.microsoft.com/en-us/library/ms978729.aspx.
- For more information on domain-driven design, see the following resources:
  - "*An Introduction To Domain-Driven Design*" at http://msdn.microsoft.com/en-us/magazine/dd419654.aspx.
  - "*Domain Driven Design and Development in Practice*" at http://www.infoq.com/articles/ddd-in-practice.

- For more information on design patterns for the business layer, see "*Service Orientation Patterns*" at http://msdn.microsoft.com/en-us/library/aa532436.aspx.
- For more information on the ADO.NET Entity Framework, see "*The ADO.NET Entity Framework Overview*" at http://msdn.microsoft.com/en-us/library/aa697427(VS.80).aspx.
- For information on business entity design with Microsoft Dynamics, see "*Business Entities*" at http://msdn.microsoft.com/en-us/library/ms940455.aspx.
- For information on business entity modeling with Microsoft Dynamics, see "*Modeling Entities*" at http://msdn.microsoft.com/en-us/library/aa475207.aspx.
- For information on using business entities with Office Business Applications (OBA), see "*Building Office Business Applications*" at http://msdn.microsoft.com/en-us/library/bb266337.aspx.
- For more information on the open source NHibernate framework, see "*NHibernate Forge*" at http://nhforge.org/Default.aspx.