**Domain and Persistence Patterns**



[**http://msdn.microsoft.com/rampup**](http://msdn.microsoft.com/rampup)

Contents

[Two Basic Types of Patterns 3](#_Toc264375441)

[Isolating Database Logic 5](#_Toc264375442)

[Data Mapper Pattern 5](#_Toc264375443)

[Repository Pattern 6](#_Toc264375444)

[Summary 7](#_Toc264375445)

# Two Basic Types of Patterns

Now we can begin to evolve our application.

There are two fundamental types of patterns: design patterns, and business logic or domain[[1]](#footnote-1) patterns.

Design patterns transcend any particular application. They might be functional, constructive, or behavioral. The factory pattern could be used, for example, where appropriate, in any application that needed to control the construction of classes or objects.

Business logic patterns reflect the type of application being built. For example, customers and orders are related in a certain way because that is how a particular company treats the relationships between customers and orders. Domain patterns emerge out of domain driven design.[[2]](#footnote-2) Since our sample app is generic, we cannot really discuss business logic patterns. Nonetheless, it is important to understand the distinction, and for you to think about patterns for your particular area of industry expertise. Nonetheless, design patterns and interface based design can be used within the domain model to partition and layer the domain model.

We can, however, describe patterns to organize the business logic. Fowler[[3]](#footnote-3) describes three architectural patterns to organize the domain logic: Transaction Script, Domain Model, and Table Module.

*Transaction Script* is probably familiar to everyone. The application takes the user input, processes the input, stores the result, and often displays some sort of result to the user. There is a one-to-one correspondence between user actions and application functionality. The code that writes to the database is usually very straight-forward.

The problem with *Transaction Script* is that as the business logic becomes more complex, the processing becomes too complicated. If the various user actions share multiple complicated processing paths, with multiple options within each path, the code logic in *Transaction Script* becomes very convoluted. Suppose adding a customer required finding out what kind of customer it is depending on credit rating, types of products they were interested in buying, their geographic area, and characteristics such as size, and their own customers. Other user actions might want to share the same functional units. Using a rich set of objects to organize the domain layer becomes much more attractive than procedural logic.

Both the *Domain Model* pattern and the *Table Module* pattern use objects, but they use those objects in a very different way. If there is a direct correspondence between the tables in the relational database and the user actions, a *Table Module* pattern can be a very attractive approach. Every class corresponds to a conceptual table in the database[[4]](#footnote-4), and a record or data set can be used to read and write to the database. Many UI widgets can work with record or data sets directly.

Not all applications have this simple relationship with database tables. Evolving the application can be more complex. When you use a Table Module pattern, you lose the full power of object oriented programming techniques. You also have a closer coupling between the business logic and the data layer. For example, as you get multiple order types, the *Table Module* pattern will have one class to handle all the orders, each order would be identified by an id.

The *Domain Model* pattern uses classes to model the entire business logic. We can then use the technique of interface based polymorphism discussed in the first unit to partition, layer, and evolve the application. In the previous order type example, each order type would be modeled with a separate class. Each order would be separate object. Hence, it would be easier to separate out different ways of handling different orders, and factor shared functionality in shared classes that use these different order types as interfaces.

The *Domain Model* does not come without a cost. Complicated object models often do not map well to relational database models. In a relational database, all the data is available to do joins and unrestricted operations. There is no encapsulation.[[5]](#footnote-5) Data is tightly coupled together by the schema. Only a limited number of operations are available. Objects have a high degree of encapsulation. With no schema, each object can define its data and behavior independently.

This can complicate query logic, and make writing ad-hoc queries difficult Object relational mapping, as will be discussed in this unit, can make this job easier, but does not remove the problem entirely.[[6]](#footnote-6)

The decision of which to use obviously depends on your application’s characteristics. We will work through the remainder of these units as if we had a domain model.

Why?

First, unless you know that your application is going to stay simple, the domain model approach will allow your application to evolve. Simple domain models will have few, if any at all, real object-relational mismatches. Changing from a *Transaction Script* or *Table Module* pattern to a *Domain Model* pattern is difficult. The code conversion can be complex, and programmers who start down the *Transaction Script* or *Table Module* path often do not understand how to use a *Domain Model* pattern. Complex domain models generally have benefits that outweigh the object relational mismatch unless end-users write a lot of ad-hoc queries. Given those benefits, for pedagogical purposes we will assume a *Domain Model* pattern. How to write a domain model is beyond the scope of this set of units. Footnote 2 has two good references on the topic.

# Isolating Database Logic

One of the most important isolation layers in an application is between the domain model code and the code that retrieves data from a data store. Usually this data store is a relational database. With this isolation, you can modify the business logic without impacting the database storage and retrieval code, and you can modify the database storage and retrieval code without changing the business logic.

The set of patterns that accomplish this task are called persistence patterns. We shall focus on two of these persistence patterns: *Data Mapper* and *Repository*. The *Data Mapper* pattern[[7]](#footnote-7) keeps the logic and database layers independent while moving data between the database and the objects. Our choice of this pattern was largely dictated by our decision to use a domain model.[[8]](#footnote-8) Object Relational Mapping (ORM) products implement the *Data Mapper* pattern. Many of them allow for object queries that can be used in a similar fashion to SQL queries. They also provide data transactions at the object level. With these and other features they can alleviate the object-relational mismatch.

## Data Mapper Pattern

To give you an idea of how the Data Mapper pattern works we have implemented a very simple object relational mapping layer in our application. To keep things simple we are not going to implement features such as lazy loading, identity maps to see if objects are already loaded, or reflection to provide complete independence between the domain model and the data mapper. An ORM can also pull in an object tree. In a real application you would use a product that provides these features rather than create your own.[[9]](#footnote-9)

Our data mapper is based on a simple interface: ICustomerDataMapper which provides just three primitive operations: delete, save, list. Save handles both creating a new instance and updating an existing instance.

public interface ICustomerDataMapper

{

void Delete(ICustomer customer);

IList<ICustomer> GetCustomers(IList<int> ids);

void Save(ICustomer customer);

}

The interface is implemented in the file CustomerDataMapper.cs. Interfaces are used as parameters to the data primitives. They are not bound to any particular implementation of a customer, or how the data stored with the customer is generated. The domain layer also has no knowledge of how the data is stored. Right now, data is stored in SortedList instances. Latter we will use a relational database with no need to change the domain model.[[10]](#footnote-10)

Notice that the CustomerManager constructor takes an ICustomerDataMapper interface:

public CustomerManager(ICustomerDataMapper dm)

This technique is called inversion of control which we will discuss in the next unit. Instead of having the CustomerManager create an instance of an object that implements the ICustomerDataMapper interface, the instance is associated with the CustomerManager. The code in the CustomerManager implementation uses this interface. The CustomerManager implementation is not bound to any particular DataMapper implementation.

## Repository Pattern

Applications need a mechanism to find existing objects. Our DataMapper layer maps objects to relational tables. Hence we need a query mechanism that queries objects, not relational tables. This is what the repository pattern does.

Our example has a very trivial example of a repository pattern to illustrate how it would work. Most ORMs implement some form of repository.

Our Criteria class represents the simple criterion used in our repository queries. We specify an operator and an operand for the query. The operators can be “All”, “Id”, or “Name” depending on whether we want all instances, query based on id or name. The operand represents the value used in the query. For example if I want the customer whose Id is 4, the operator would be “Id” and the operand would be “4”.

public class Criteria

{

public string Operator { get; set; }

public object Operand { get; set; }

}

The Repository’s Query method returns a collection of objects.

The Repository uses the DataMapper in its implementation. Again, an interface is passed into the Customer Repository constructor using inversion of control.

public CustomerRepository(ICustomerDataMapper dm)

Since we are using interface based design, it makes it easy to replace our SortedList based implementation with a relational based implementation of the DataMapper.

Using the repository pattern allows developers to construct queries in terms of the objects they are using, without having to understand the details of how the database is implemented or the intracies of whatever SQL dialect the database uses. Even in our simplistic implementation, asking for a customer brings back the related order. The repository pattern allows the developer to think in terms of the domain objects with which they are programming. They get back a fully instantiated chain of the objects they need without having to worry about how to reconstitute them from the data store implementation.

# Summary

While domain patterns are industry specific, the idea of separating the business logic from the way data is stored in the database is fundamental.

We have used interface based design to look at two key persistence patterns: *DataMapper* and *Repository*. While these are not the only two persistence patterns, they illustrate two critical elements in any data hiding layer: hiding the details of the database implementation, and providing data access in terms of the programming concepts and abstractions used in the business logic layer.

1. Here is a good example of how words are overloaded in the software literature. Domain driven design refers to determining the proper abstractions to use in a software application. These abstractions are often called the domain model. Fowler (see footnote 3), uses the term domain model pattern or layer to refer to where a complex set of objects is used to implement the domain model. As we will discuss shortly, there are other ways to implement the domain model. Fowler distinguishes between Domain Logic and Domain Model pattern which can be quite confusing. [↑](#footnote-ref-1)
2. Eric Evans, Domain Driven Design, and Jimmy Nilsson, Applying Domain-Driven Design and Patterns are two good books that discuss domain driven design. [↑](#footnote-ref-2)
3. Martin Fowler, Patterns of Enterprise Application Architecture, Chapters 2 and 9 [↑](#footnote-ref-3)
4. A conceptual table means that the business logic code views the data from the database as a single table. The data coming back or being stored might actually be the result of a join, or some other database operation. [↑](#footnote-ref-4)
5. Views are not used for programming language style encapsulation. [↑](#footnote-ref-5)
6. See Ted Neward’s famous blog post on [ORM as the Vietnam of Computer Science](http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx) for a good discussion of the problem even if you find his analogy overdone. [↑](#footnote-ref-6)
7. This is Fowler’s name for the pattern. [↑](#footnote-ref-7)
8. As noted in the last section, this is a simulation. [↑](#footnote-ref-8)
9. See <http://ayende.com/Blog/archive/2006/05/12/25ReasonsNotToWriteYourOwnObjectRelationalMapper.aspx>

   for a good explanation why you would never write your own ORM. [↑](#footnote-ref-9)
10. Fowler has a more detailed example of a DataMapper. [↑](#footnote-ref-10)