08/17/2015 • 17 minutes to read

In this article

August 2009

Volume 24 Number 08

Cutting Edge - Pros and Cons of Data Transfer Objects

By Dino Esposito | August 2009

Contents

Nearly every developer and architect would agree on the following, though relatively loose, definition of the business logic layer (BLL) of a software application: The BLL is the part of the software application that deals with the performance of business-related tasks. Code in the BLL operates on data that attempts to model entities in the problem domain—invoices, customers, orders, and the like. Operations in the BLL attempt to model business processes.

Under the hood of this largely accepted definition lie a number of key details that are left undefined and unspecified. Design patterns exist to help architects and code designers transform loose definitions into blueprints. In general, BLL design patterns have a slightly different focus. They model operations and data and often serve as the starting point for designing the BLL.

In this article, after a brief refresher on procedural and object-based patterns for organizing the BLL, I'll focus on one side of the problem—data transfer objects—that if not effectively addressed at the architecture level, may have a deep impact on the development of the project.

Procedural Patterns for BLL

When it comes to designing the BLL, you can start from the use-cases that have emerged during the analysis phase. Typically, you end up coding one method for each required interaction between the user and the system. Each interaction forms a logical transaction that includes all due steps—from collecting input to performing the task, and from database access to refreshing the user interface. This approach is referred to as the Transaction Script (TS) pattern.

In TS, you focus on the required actions and don't really build a conceptual model of the domain as the gravitational center of the application.

To move data around, you can use any container objects that may suit you. In the Microsoft .NET space, this mostly means using ADO.NET data containers such as DataSets and DataTables. These objects are a type of super-array object, with some search, indexing, and filtering capabilities. In addition, DataSets and DataTables can be easily serialized across tiers and even persisted locally to enable offline scenarios.

The TS pattern doesn't mandate a particular model for data representation (and doesn't prevent any either). Typically, operations are grouped as static methods in one or more entry-point classes. Alternatively, operations can be implemented

as commands in a Command pattern approach. Organization of data access is left to the developer and typically results in chunks of ADO.NET code.

The TS pattern is fairly simple to set up; at the same time, it obviously doesn't scale that well, as the complexity of the application grows. In the .NET space, another pattern has gained wide acceptance over the years: the Table Module pattern. In a nutshell, the Table Module pattern suggests a database-centric vision of the BLL. It requires you to create a business component for each database table. Known as the table module class, the business component packages the data and behavior together.

In the Table Module pattern, the BLL is broken into a set of coarse-grained components, each representing an entire database table. Being strictly table-oriented, the Table Module pattern lends itself to using recordset-like data structures for passing data around. ADO.NET data containers or, better yet, customized and typed version of ADO.NET containers are the natural choice.

As the need for a more conceptual view of the problem domain arises, the BLL patterns that have worked for years in the .NET space need to evolve some more. Architects tend to build an entity/relationship model that represents the problem domain and then look at technologies like LINQ-to-SQL and Entity Framework as concrete tools to help.

Object-Based Patterns for BLL

The Table Module pattern is based on objects, but it's not really an object-based pattern for modeling the business logic. It does have objects, but they are objects representing tables, not objects representing the domain of the problem.

In an object-oriented design, the business logic identifies entities and expresses all of the allowed and required interactions between entities. In the end, the application is viewed as a set of interrelated and interoperating objects. The set of objects mapping to entities, plus some special objects performing calculations form the domain model. (In the Entity Framework, you express the domain model using the Entity Data Model [EDM].)

There are various levels of complexity in a domain model that suggest different patterns—typically the Active Record pattern or the Domain Model pattern. A good measure of this complexity is the gap between the entity model you have in mind and the relational data model you intend to create to store data. A simple domain model is one in which your entities map closely

to tables in the data model. A not-so-simple model requires mapping to load and save domain objects to a relational database.

The Active Record pattern is an ideal choice when you need a simple domain model; otherwise, when it is preferable to devise entities and relationships regardless of any database notion, the Domain Model pattern is the way to go.

The Active Record pattern is similar to what you get from a LINQ-to-SQL object model (and the defaultgenerated model with the Entity Designer in the Entity Framework Version 1.0). Starting from an existing database, you create objects that map a row in a database table. The object will have one property for each table column of the same type and with the same constraints. The original formulation of the Active Record pattern recommends that each object makes itself responsible for its own persistence. This means that each entity class should include methods such as Save and Load. Neither LINQ-to-SQL nor Entity Framework does this though, as both delegate persistence to an integrated O/RM infrastructure that acts as the real data access layer, as shown in **Figure 1**.

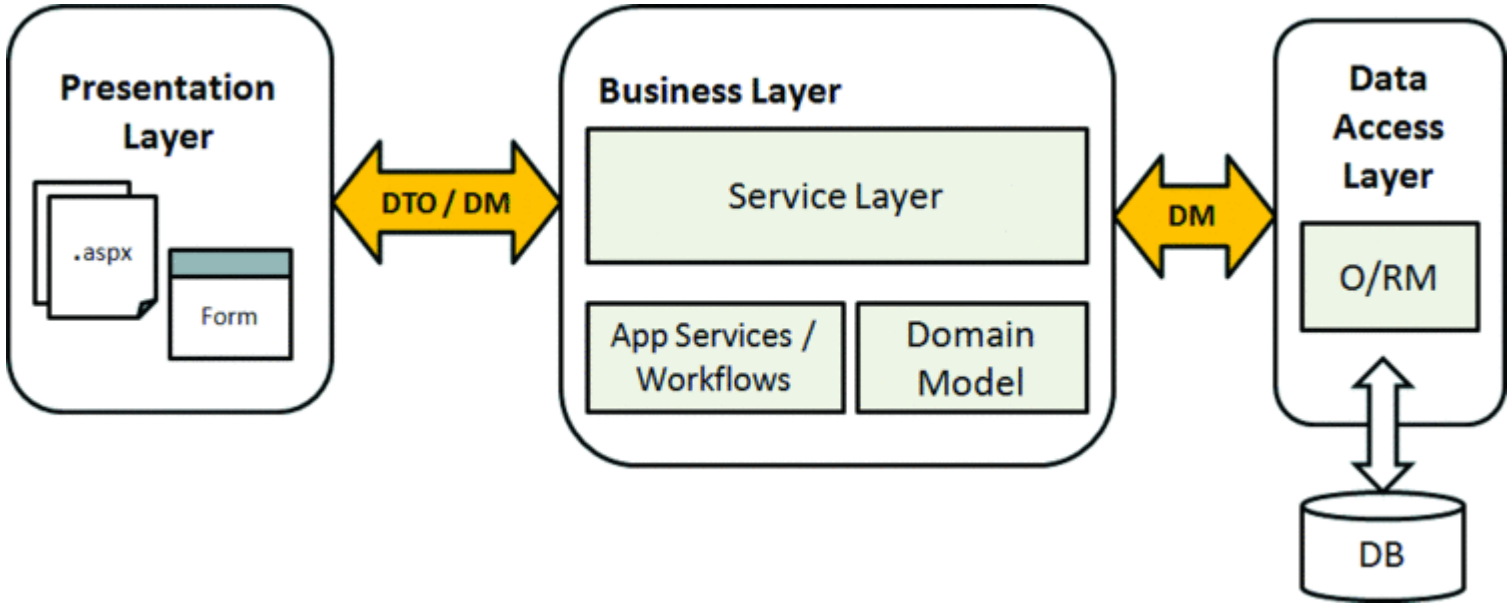

Figure 1 **– the Domain Model Pattern Used for the BLL**

The Service Layer

In **Figure 1**, you see a logical section of the BLL named as the "service layer" sitting in between the presentation layer and the layer that takes care of persistence. In a nutshell, the service layer defines an interface for the presentation layer to trigger predefined system actions. The service layer decouples presentation and business logic and represents the façade for the presentation logic to call into the BLL. The service layer does its own job, regardless of how the business logic is organized internally.

As a .NET developer, you are quite familiar with event handlers in Web or Windows forms. The canonical Button1_Click method belongs to the presentation layer and expresses the system's behavior after the user has clicked a given button. The system's behavior—more exactly, the use case you're implementing—may require some interaction with BLL components. Typically, you need to instantiate the BLL component and then script it. The code necessary to script the component may be as simple as calling the constructor and perhaps one method. More often, though, such code is fairly rich with branches, and may need to call into multiple objects or wait for a response. Most developers refer to this code as application logic. Therefore, the service layer is the place in the BLL where you store application logic, while keeping it distinct and separate from domain logic. The domain logic is any logic you fold into the classes that represent domain entities.

In **Figure 1**, the service layer and domain model blocks are distinct pieces of the BLL, although they likely belong to different assemblies. The service layer knows the domain model and references the corresponding assembly. The service layer assembly, instead, is referenced from the presentation layer and represents the only point of contact between any presentation layer (be it Windows, Web, Silverlight, or mobile) and the BLL. **Figure 2** shows the graph of references that connect the various actors. The service layer is a sort of mediator between the presentation layer and the rest of the BLL. As such, it keeps them neatly separated but loosely coupled so that they are perfectly able to communicate. In **Figure 2**, the presentation layer doesn't hold any reference to the domain model assembly. This is a key design choice for most layered solutions.

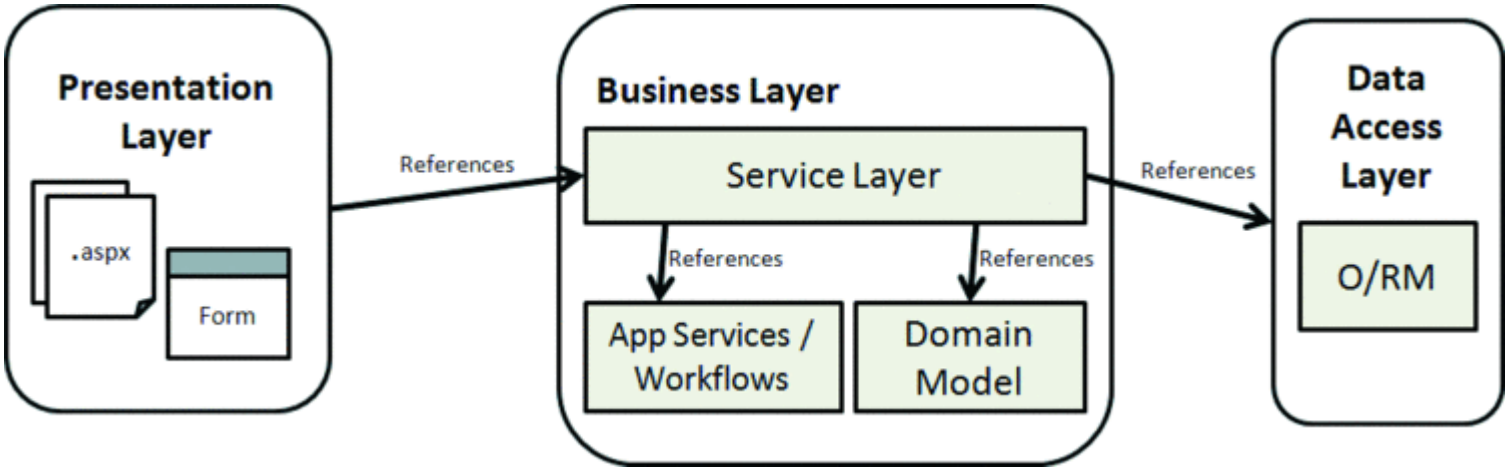

Figure 2

Introducing Data Transfer Objects

When you have a domain-based vision of the application, you can't help but look seriously into data transfer objects. No multitier solution based on LINQ to SQL or Entity Framework is immune from this design issue. The question is, how would you move data to and from the presentation layer? Put another way, should the presentation layer hold a reference to the domain model assembly? (In an Entity Framework scenario, the domain model assembly is just the DLL created out of the EDMX file.)

Ideally, the design should look like **Figure 3**, where made-to-measure objects are used to pass data from the presentation layer to the service layer, and back. These ad hoc container objects take the name of Data Transfer Objects (DTOs).

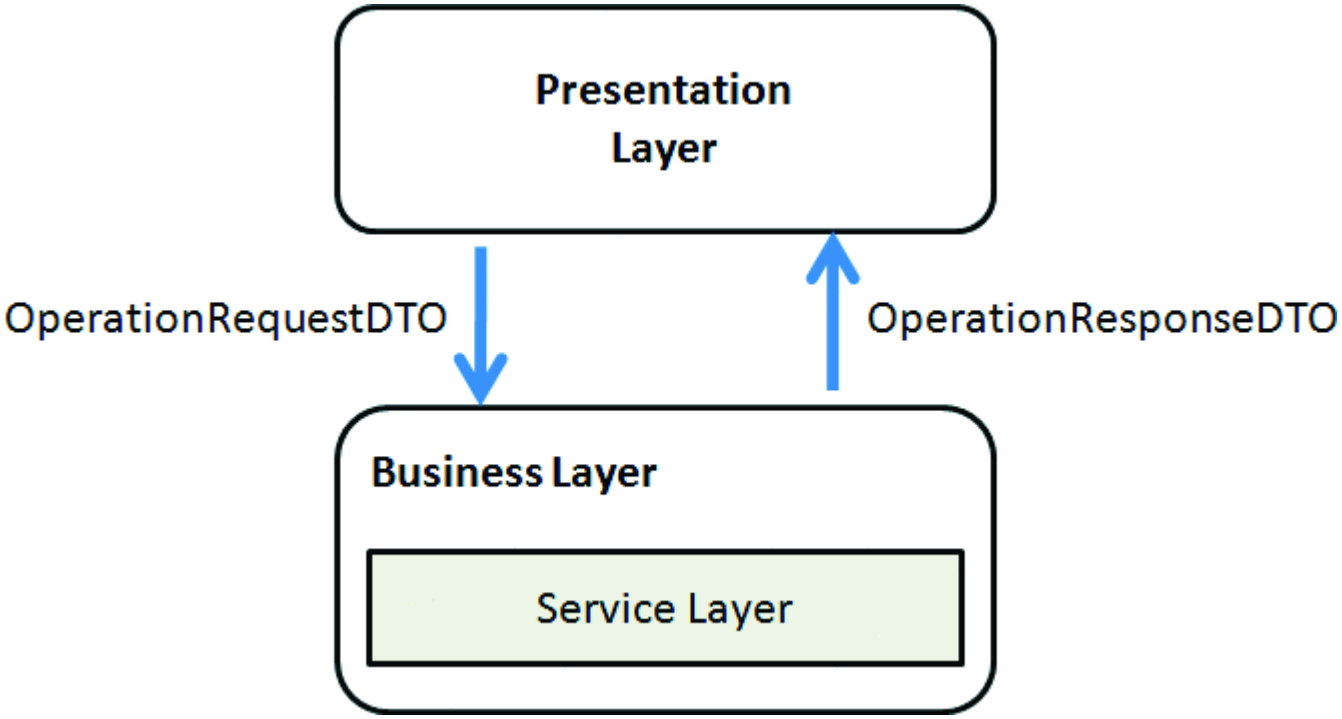

Figure 3

A DTO is nothing more than a container class that exposes properties but no methods. A DTO is helpful whenever you need to group values in ad hoc structures for passing data around.

From a pure design perspective, DTOs are a solution really close to perfection. DTOs help to further decouple presentation from the service layer and the domain model. When DTOs are used, the presentation layer and the service layer share data contracts rather than classes. A data contract is essentially a neutral representation of the data that interacting components exchange. The data contract describes the data a component receives, but it is not a system-specific class, like an entity. At the end of the day, a data contract is a class, but it is more like a helper class specifically created for a particular service method.

A layer of DTOs isolates the domain model from the presentation, resulting in both loose coupling and optimized data transfer.

Other Benefits of DTOs

The adoption of data contracts adds a good deal of flexibility to the service layer and subsequently to the design of the entire application. For example, if DTOs are used, a change in the requirements that forces a move to a different amount of data doesn't have any impact on the service layer or even the domain. You modify the DTO class involved by adding a new property, but leave the overall interface of the service layer intact.

It should be noted that a change in the presentation likely means a change in one of the use cases and therefore in the application logic. Because the service layer renders the application logic, in this context a change in the service layer interface is still acceptable. However, in my experience, repeated edits to the service layer interface may lead to the wrong conclusion that changes in the domain objects—the entities—may save you further edits in the service layer. This doesn't happen in well-disciplined teams or when developers have a deep understanding of the separation of roles that exists between the domain model, the service layer, and DTOs.

As **Figure 4** shows, when DTOs are employed, you also need a DTO adapter layer to adapt one or more entity objects to a different interface as required by the use case. In doing so, you actually implement the "Adapter" pattern—one of the classic and most popular design patterns. The Adapter pattern essentially converts the interface of one class into another interface that a client expects.

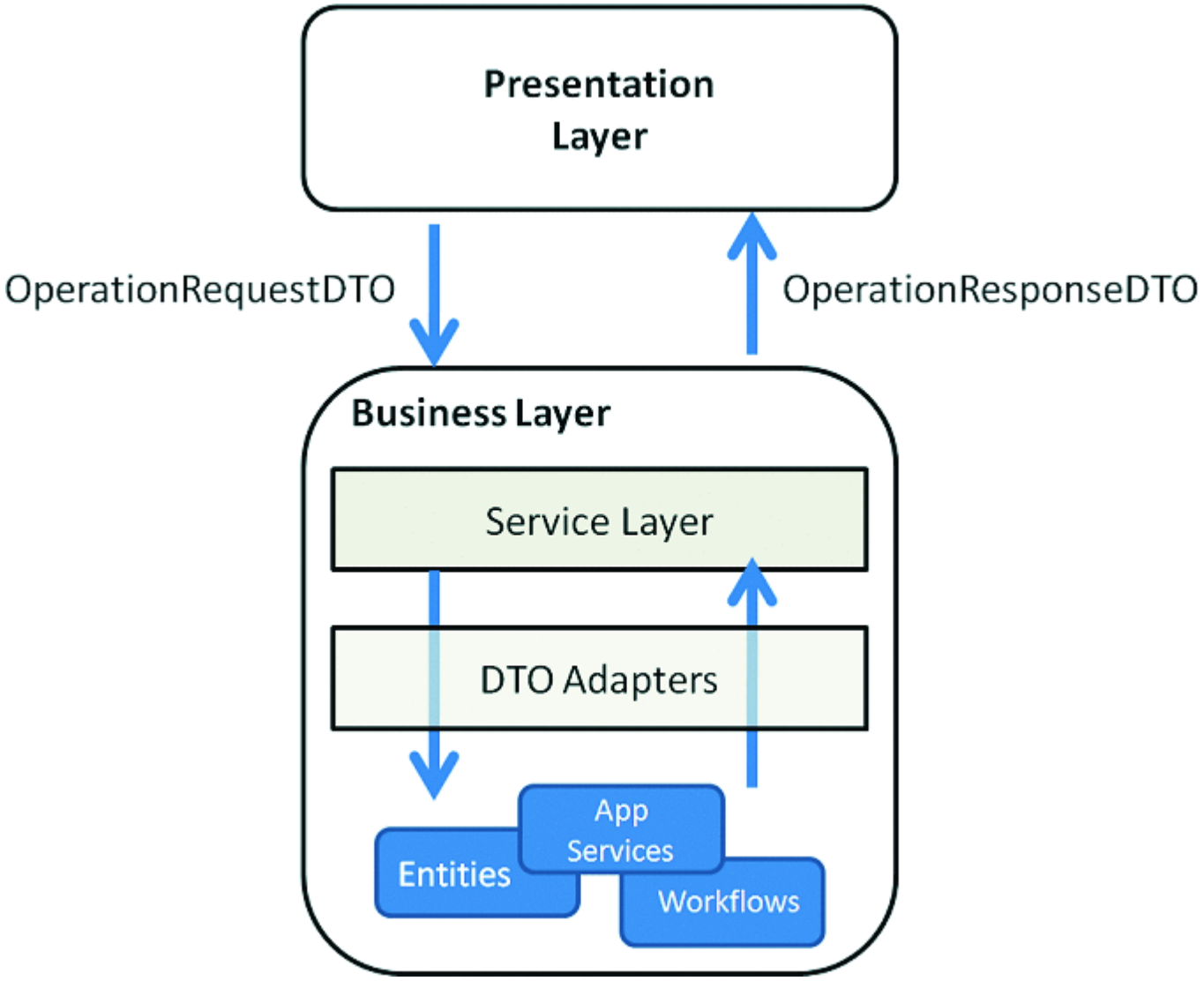

Figure 4

With reference to **Figure 4**, the adapter layer is responsible for reading an incoming instance of the OperationRequestDTO class and for creating and populating fresh instances of OperationResponseDTO.

When requirements change and force changes in a DTO-based service layer, all you need to do is update the public data contract of the DTO and adjust the corresponding DTO adapter.

The decoupling benefits of DTOs don't end here. In addition, to happily surviving changes in the presentation, you can enter changes to the entities in the domain model without impacting any clients you may have.

Any realistic domain model contains relationships, such as Customer-to-Orders and Order-to-Customer, that form a double link between Customer and Order entities. With DTOs, you also work around the problem of managing circular references during the serialization of entity objects. DTOs can be created to carry a flat stream of values that, if needed, serialize just fine across any boundaries. (I'll return to this point in a moment.)

Drawbacks of DTOs

From a pure design perspective, DTOs are a real benefit, but is this theoretical point confirmed by practice, too? As in many architecture open points, the answer is, it depends.

Having hundreds of entities in the domain model is definitely a good reason for considering alternatives to a pure DTO-based approach. In large projects with so many entities, DTOs add a remarkable level of (extra) complexity and work to do. In short, a pure, 100% DTO solution is often just a 100 percent painful solution.

While normally the complexity added to a solution by DTOs is measured with the cardinality of the domain model, the real number of needed DTOs can be more reliably determined looking at the use cases and the implementation of the service layer. A good formula for estimating how many DTOs you need is to look at the number of methods in the service layer. The real number can be smaller if you are able to reuse some DTOs across multiple service layer calls, or higher if your DTOs group some data using complex types.

In summary, the only argument against using DTOs is the additional work required to write and manage the number of resulting DTO classes. It is not, however, a simple matter of a programmer's laziness. In large projects, decoupling presentation from the service layer costs you hundreds of new classes.

It should also be noted that a DTO is not simply a lightweight copy of every entity you may have. Suppose that two distinct use cases require you to return a collection of orders—say, GetOrdersByCountry and GetOrdersByCustomer. Quite likely, the information to put in the "order" is different. You probably need more (or less) details in GetOrdersByCustomer than in GetOrdersByCountry. This means that distinct DTOs are necessary. For this reason, hundreds of entities are certainly a quick measure of complexity, but the real number of DTOs can be determined only by looking at use cases.

If DTOs are not always optimal, what would be a viable alternate approach?

The only alternative to using DTOs is to reference the domain model assembly from within the presentation layer. In this way though, you establish a tight coupling between layers. And tightly coupled layers may be an even worse problem.

Referencing Entities Directly

A first, not-so-obvious condition to enable the link of entities directly from the presentation layer is that it is acceptable for the presentation layer to receive data in the format of entity objects. Sometimes the presentation needs data formatted in a particular manner. A DTO adapter layer exists to just massage data as required by the client. If you don't use DTOs though, the burden of formatting data properly must be moved onto the presentation layer. In fact, the wrong place in which to format data for user interface purposes is the domain model itself.

Realistically, you can do without DTOs only if the presentation layer and the service layer are co-located in the same process. In this case, you can easily reference the entity assembly from within both layers without dealing with thorny issues such as remoting and data serialization. This consideration leads to another good question: Where should you fit the service layer?

If the client is a Web page, the service layer is preferably local to the Web server that hosts the page. In ASP.NET applications, the presentation layer is all in code-behind classes and lives side by side with the service layer in the same AppDomain. In such a scenario, every communication between the presentation layer and the service layer occurs in-process and objects can be shared with no further worries. ASP.NET applications are a good scenario where you can try a solution that doesn't use the additional layer of DTOs.

Technology-wise, you can implement the service layer via plain .NET objects or via local Windows Communication Foundation (WCF) services. If the application is successful, you can easily increase scalability by relocating the service layer to a separate application server.

If the client is a desktop application, then the service layer is typically deployed to a different tier and accessed remotely from the client. As long as both the client and remote server share the same .NET platform, you can use remoting techniques (or, better, WCF services) to implement communication and still use native entity objects on both ends. The WCF infrastructure will take care of marshaling data across tiers and pump it into copies of native entities. Also, in this case you can arrange an architecture that doesn't use DTOs. Things change significantly if the client and server platforms are incompatible. In this

case, you have no chances to link the native objects and invoke them from the client; subsequently, you are in a pure service-oriented scenario and using DTOs is the only possibility.

The Middle Way

DTOs are the subject of an important design choice that affects the implementation of any communication between the presentation and the back end of the system.

If you employ DTOs, you keep the system loosely coupled and open toward a variety of clients. DTOs are the ideal choice, if you can afford it. DTOs add a significant programming overhead to any real-world system. This doesn't mean that DTOs should not be used, but they lead to a proliferation of classes that can really prefigure a maintenance nightmare in projects with a few hundred entity objects and even more use cases.

If you are at the same time a provider and consumer of the service layer, and if you have full control over the presentation, there might be benefits in referencing the entity model assembly from the presentation. In this way, all methods in the service layer are allowed to use entity classes as the data contracts of their signatures. The impact on design and coding is clearly quite softer.

Whether to use DTOs or not is not a point easy to generalize. To be effective, the final decision should always be made looking at the particulars of the project. In the end, a mixed approach is probably what you'll be doing most of the time. Personally, I tend to use entities as much as I can. This happens not because I'm against purity and clean design, but for a simpler matter of pragmatism. With an entity model that accounts for only 10 entities and a few use cases, using DTOs all the way through doesn't pose any significant problem. And you get neat design and low coupling. However, with hundreds of entities and use cases, the real number of classes to write, maintain, and test ominously approaches the order of thousands. Any possible reduction of complexity that fulfills requirements is more than welcome.

As an architect, however, you should always be on the alert to recognize signs indicating that the distance between the entity model and what the presentation expects is significant or impossible to cover. In this case, you should take the safer (and cleaner) route of DTOs.

Mixed Approach

Today's layered applications reserve a section of the BLL to the service layer. The service layer (also referred to as the application layer) contains the application logic; that is, the business rules and procedures that are specific to the application but not to the domain. A system with multiple front ends will expose a single piece of domain logic through entity classes, but then each front end will have an additional business layer specific to the use cases it supports. This is what is referred to as the service (or application) layer.

Triggered from the UI, the application logic scripts the entities and services in the business logic. In the service layer, you implement the use cases and expose each sequence of steps through a coarse-grained method for the presentation to call.

In the design of the service layer, you might want to apply a few best practices, embrace service-orientation, and share data contracts instead of entity classes. While this approach is ideal in theory, it often clashes with the real world, as it ends up adding too much overhead in projects with hundreds of entities and use cases.

It turns out that a mixed approach that uses data contracts only when using classes is not possible, is often the more acceptable solution. But as an architect, you must not make this decision lightly. Violating good design rules is allowed, as long as you know what you're doing.

Send your questions and comments for Dino to cutting@microsoft.com.

Dino Esposito is an architect at IDesign and co-author of *Microsoft .NET: Architecting Applications for the Enterprise* (Microsoft Press, 2008). Based in Italy, Dino is a frequent speaker at industry events worldwide. You can join his blog at weblogs.asp.net/despos.