Module 1

Clientele Training

Table of Contents

[Preface 2](#_Toc357759323)

[Introduction 3](#_Toc357759324)

[Architecture 3](#_Toc357759325)

[Enterprise Applications 3](#_Toc357759326)

[Kinds of Enterprise Application 4](#_Toc357759327)

[Thinking About Performance 5](#_Toc357759328)

[Layering 7](#_Toc357759329)

[The Evolution of Layers in Enterprise Applications 7](#_Toc357759330)

[The Three Principal Layers 8](#_Toc357759331)

[Choosing Where to Run Your Layers 10](#_Toc357759332)

[Organizing Domain Logic 12](#_Toc357759333)

[Making a Choice 14](#_Toc357759334)

[Service Layer 15](#_Toc357759335)

[Mapping to Relational Databases 17](#_Toc357759336)

[Architectural Patterns 17](#_Toc357759337)

[Table Data Gateway 17](#_Toc357759338)

[Data Mapper 17](#_Toc357759339)

[Repository 18](#_Toc357759340)

[Query Object 18](#_Toc357759341)

[Object Relational Mapper 18](#_Toc357759342)

[Active Record 18](#_Toc357759343)

[Memento 18](#_Toc357759344)

[The Behavioural Problem 19](#_Toc357759345)

# Preface

The information contained in this document is sourced from Martin Fowlers book [Patterns of Enterprise Application Architecture](http://www.amazon.com/Patterns-Enterprise-Application-Architecture-Martin/dp/0321127420). This is intended as supplementary information for the Clientele Advanced Training and should not be distributed.

# Introduction

## Architecture

"Architecture" is a term that lots of people try to define, with little agreement. There are two common elements: One is the highest-level breakdown of a system into its parts; the other, decisions that are hard to change.

Architecture is a subjective thing, a shared understanding of a system's design by the expert developers on a project. Commonly this shared understanding is in the form of the major components of the system and how they interact.

## Enterprise Applications

Enterprise applications include payroll, patient records, shipping tracking, cost analysis, credit scoring, insurance, supply chain, accounting, customer service, and foreign exchange trading.

Enterprise applications don't include automobile fuel injection, word processors, elevator controllers, chemical plant controllers, telephone switches, operating systems, compilers, and games.

Enterprise applications usually involve persistent data. **Over time there will be many changes to the structure of the data in order to store new pieces of information without disturbing the old pieces.**

**Usually many people access data concurrently**. For many systems this may be less than a hundred people, but for Web-based systems that talk over the Internet this goes up by orders of magnitude. With so many people there are definite issues in ensuring that all of them can access the system properly.

With so much data, there are usually a lot of user interface screens to handle it. Users of enterprise applications vary from occasional to regular, and normally they will have little technical expertise. Thus, the **data has to be presented lots of different ways for different purposes**.

**Enterprise applications usually need to integrate** **with other enterprise applications** scattered around the enterprise. The various systems are built at different times with different technologies, and even the collaboration mechanisms will be different.

**There are often problems with differences in business process and conceptual dissonance with the data.** One division of the company may think a customer is someone with whom it has a current agreement; another division also counts those that had a contract but don't any longer; another counts product sales but not service sales.

**Business logic may often be entirely illogical**. You have to deal with a haphazard array of strange conditions that often interact with each other in surprising ways. Of course, they got that way for a reason: Some salesman negotiated to have a certain yearly payment two days later than usual because that fit with his customer's accounting cycle and thus won a couple of million dollars in business. A few thousand of these one-off special cases is what leads to the complex business "illogic" that makes business software so difficult. The only certain thing is that the logic will change over time.

For some people the term "enterprise application" implies a large system. However, it's important to remember that not all enterprise applications are large. Such thinking tends to short-change the cumulative effect of many small projects. The cumulative effect of small projects can be very significant on an enterprise, particularly since they often have disproportionate value. Indeed, one of the best things you can do is turn a large project into a small one by simplifying its architecture and process.

## Kinds of Enterprise Application

It is important to realize that enterprise applications are all different and that different problems lead to different ways of doing things. I have a set of alarm bells that go off when people say, "Always do this."

Consider a B2C (business to customer) online retailer. For such a system we need to be able to handle a very high volume of users, so our solution needs to be not only reasonably efficient in terms of resources used but also scalable so that you can increase the load by adding more hardware. The domain logic for such an application can be pretty straightforward. We want anyone to be able access the system easily, so that implies a pretty generic Web presentation that can be used with the widest possible range of browsers. Data source includes a database for holding orders and perhaps some communication with an inventory system to help with availability and delivery information.

Contrast this with a system that automates the processing of leasing agreements. There are many fewer users; no more than a hundred or so at one time. Where it's more complicated is in the business logic. Calculating monthly bills on a lease, handling events such as early returns and late payments, and validating data as a lease is booked are all complicated tasks, since much of the leasing industry's competition comes in the form of little variations over deals done in the past. A complex business domain such as this is challenging because the rules are so arbitrary.

Such a system also has more complexity in the user interface (UI). At the least this means a much more involved HTML interface with more, and more complex, screens. Often these systems have UI demands that lead users to want a more sophisticated presentation than a HTML front end allows, so a more conventional rich-client interface is needed.

A third example point is a simple expense-tracking system for a small company. Such a system has few users and simple logic and can easily be made accessible across the company with an HTML presentation. The only data source is a few tables in a database. As simple as it is, a system like this is not devoid of a challenge. You have to build it very quickly and you have to bear in mind that it may grow in complexity over time. Trying to use the architecture for either of the other two example systems will slow down the development of this one. Although such systems may be small, most enterprises have a lot of them so the cumulative effect of an inappropriate architecture can be significant.

Each of these three enterprise application examples has difficulties, and they are different difficulties. As a result you can't come up with a single architecture that will be right for all three. Choosing an architecture means that you have to understand the particular problems of your system and choose an appropriate design based on that understanding. That is why there is no single solution for your enterprise needs. Instead, many of the patterns are about choices and alternatives. Even when you choose a particular pattern, you'll have to modify it to meet your demands. You can't build enterprise software without thinking, and all any book can do is give you more information to base your decisions on.

If this applies to patterns, it also applies to tools. Although it obviously makes sense to pick as small a set of tools as you can to develop applications, you also have to recognize that different tools are best for different purposes.

## Thinking About Performance

Many architectural decisions are about performance. For most performance issues I prefer to get a system up and running, instrument it, and then use a disciplined optimization process based on measurement. However, some architectural decisions affect performance in a way that's difficult to fix with later optimization.

Any advice on performance should not be treated as fact until it's measured on your configuration. Too often I've seen designs used or rejected because of performance considerations, which turn out to be bogus once somebody actually does some measurements on the real setup used for the application.

Whenever you do a performance optimization you must measure both before and after, otherwise, you may just be making your code harder to read.

Another problem with talking about performance is the fact that many terms are used in an inconsistent way. The most noted victim of this is "scalability," which is regularly used to mean half a dozen different things.

Here are the terms I use.

**Response time** is the amount of time it takes for the system to process a request from the outside. This may be a UI action, such as pressing a button, or a server API call.

**Responsiveness** is about how quickly the system acknowledges a request as opposed to processing it. This is important in many systems because users may become frustrated if a system has low responsiveness, even if its response time is good. If your system waits during the whole request, then your responsiveness and response time are the same. However, if you indicate that you've received the request before you complete, then your responsiveness is better

**Latency** is the minimum time required to get any form of response, even if the work to be done is non-existent. It's usually the big issue in remote systems because of the time taken for the request and response to make their way across the wire. Latency is also the reason why you should minimize remote calls.

**Throughput** is how much stuff you can do in a given amount of time. For enterprise applications a typical measure is transactions per second (tps).

**Load** is a statement of how much stress a system is under, which might be measured in how many users are currently connected to it.

**Load sensitivity** is an expression of how the response time varies with the load.

**Efficiency** is performance divided by resources. A system that gets 30 tps on two CPUs is more efficient than a system that gets 40 tps on four identical CPUs.

**Capacity** of a system is an indication of maximum effective throughput or load. This might be an absolute maximum or a point at which the performance dips below an acceptable threshold.

**Scalability** is a measure of how adding resources (usually hardware) affects performance. A scalable system is one that allows you to add hardware and get a commensurate performance improvement, such as doubling how many servers you have to double your throughput. Vertical scalability, or scaling up, means adding more power to a single server, such as more memory. Horizontal scalability, or scaling out, means adding more servers.

When building enterprise systems, it often makes sense to build for hardware scalability rather than capacity or even efficiency. Scalability gives you the option of better performance if you need it. Scalability can also be easier to do. Often designers do complicated things that improve the capacity on a particular hardware platform when it might actually be cheaper to buy more hardware. It is often cheaper to add more servers than it is to add more programmers; providing that a system is scalable.

# Layering

Layering is one of the most common techniques that software designers use to break apart a complicated software system. When thinking of a system in terms of layers, you imagine the principal subsystems in the software arranged in some form of layer cake, where each layer rests on a lower layer. In this scheme the higher layer uses various services defined by the lower layer, but the lower layer is unaware of the higher layer.

Breaking down a system into layers has a number of important benefits.

* You can understand a single layer as a coherent whole without knowing much about the other layers.
* You can substitute layers with alternative implementations of the same basic services.
* You minimize dependencies between layers
* Layers make good places for standardization.
* Once you have a layer built, you can use it for many higher-level services

Layering is an important technique, but there are downsides.

* Layers encapsulate some, but not all, things well. As a result you sometimes get cascading changes.
* Extra layers can harm performance. At every layer things typically need to be transformed from one representation to another. However, the encapsulation of an underlying function often gives you efficiency gains that more than compensate.

But the hardest part of a layered architecture is deciding what layers to have and what the responsibility of each layer should be.

## The Evolution of Layers in Enterprise Applications

In the early days of batch systems you wrote a program that manipulated some form of files and that was your application. No layers need apply.

The notion of layers became more apparent in the '90s with the rise of client-server systems. These were two-layer systems: The client held the user interface and other application code, and the server was usually a relational database.

Common client tools were VB, Powerbuilder, and Delphi. These made it particularly easy to build data-intensive applications, as they had UI widgets that were aware of SQL. Thus you could build a screen by dragging controls onto a design area and then using property sheets to connect the controls to the database. (MVC provides this functionality through its ‘scaffolding’ tooling)

If the application was all about the display and simple update of relational data, then these client-server systems worked very well. The problem came with domain logic: business rules, validations, calculations, and the like. Usually people would write these on the client, but this was awkward and usually done by embedding the logic directly into the UI screens. As the domain logic got more complex, this code became very difficult to work with. Furthermore, embedding logic in screens made it easy to duplicate code, which meant that simple changes resulted in hunting down similar code in many screens.

An alternative was to put the domain logic in the database as stored procedures. However, stored procedures gave limited structuring mechanisms, which again led to awkward code. Also, many people liked relational databases because SQL was a standard that would allow them to change their database vendor. Despite the fact that few people actually did this, many liked having the option to change vendors without too high a porting cost. Because they are all proprietary, stored procedures removed that option.

At the same time that client-server was gaining popularity, the object-oriented world was rising. The object community had an answer to the problem of domain logic: Move to a three-layer system. In this approach you have a presentation layer for your UI, a domain layer for your domain logic, and a data source. This way you could move all of that intricate domain logic out of the UI and put it into a layer where you could structure it properly with objects.

Despite this, the object bandwagon made little headway. The truth was that many systems were simple, or at least started that way. And although the three-layer approach had many benefits, the tooling for client-server was compelling if your problem was simple. The client-server tools also were difficult, or even impossible, to use in a three-layer configuration.

I think the seismic shock here was the rise of the Web. Suddenly people wanted to deploy client-server applications with a Web browser. However, if all your business logic was buried in a rich client, then all your business logic needed to be redone to have a Web interface. A well-designed three-layer system could just add a new presentation layer and be done with it. Furthermore, with Java we saw an unashamedly object-oriented language hit the mainstream. The tools that appeared to build Web pages were much less tied to SQL and thus more amenable to a third layer.

When people discuss layering, there's often some confusion over the terms layer and tier. Often the two are used as synonyms, but most people see tier as implying a physical separation. Client-server systems are often described as two-tier systems, and the separation is physical: The client is a desktop and the server is a server.

I use layer to stress that you don't have to run the layers on different machines. A distinct layer of domain logic often runs on either a desktop or the database server. In this situation you have two nodes but three distinct layers. With a local database I can run all three layers on a single laptop, but there will still be three distinct layers.

## The Three Principal Layers

For this book I'm centring my discussion around an architecture of three primary layers: presentation, domain, and data source.

**Presentation logic** *(also referred to as the user interface)* is about how to handle the interaction between the user and the software. The primary responsibilities of the presentation layer are to display information to the user and to interpret commands from the user into actions upon the domain and data source.

**Data source logic** *(also referred to* *as the infrastructure or persistence layer)* is about communicating with other systems that carry out tasks on behalf of the application. These can be transaction monitors, other applications, messaging systems, databases and so forth.

**The domain logic** *(also referred to as the business layer)*. Is about performing the work that this application needs to do for the domain you're working with. It involves calculations based on inputs and stored data, validation of any data that comes in from the presentation, and figuring out exactly what data source logic to dispatch, depending on commands received from the presentation.

So far I've talked about a user. This naturally raises the question of what happens when there is no a human being driving the software. This could be something new and fashionable like a Web service or something mundane and useful like a batch process. In the latter case the user is the client program.

At this point it becomes apparent that there is a lot of similarity between the presentation and data source layers in that they both are about connection to the outside world. This is the logic behind Alistair Cockburn's Hexagonal Architecture pattern [wiki], which visualizes any system as a core surrounded by interfaces to external systems. In Hexagonal Architecture everything external is fundamentally an outside interface, and thus it's a symmetrical view rather than my asymmetric layering scheme.

However, asymmetry is useful because there is a good distinction to be made between an interface that you provide as a service to others and your use of someone else's service. Driving down to the core, this is the real distinction between presentation and data source. Presentation is an external interface for a service your system offers to someone else, whether it is a human or a simple remote program. Data source is the interface to things that are providing a service to you. I find it beneficial to think about these differently because the difference in clients alters the way you think about the service.

Although we can identify the three common responsibility layers of presentation, domain, and data source for every enterprise application, how you separate them depends on how complex the application is. A simple script to pull data from a database and display it in a Web page may all be one procedure. I would still endeavour to separate the three layers, but in that case I might do it only by placing the behaviour of each layer in separate subroutines. As the system gets more complex, I would break the three layers into separate classes.

As complexity increased I would divide the classes into separate projects. My general advice is to choose the most appropriate form of separation for your problem but make sure you do some kind of separation, at least at the subroutine level.

Together with the separation, there's also a steady rule about dependencies: The domain and data source should never be dependent on the presentation. That is, there should be no subroutine call from the domain or data source code into the presentation code. This rule makes it easier to substitute different presentations on the same foundation and makes it easier to modify the presentation without serious ramifications deeper down. The relationship between the domain and the data source is more complex and depends upon the architectural patterns used for the data source.

One of the hardest parts of working with domain logic seems to be that people often find it difficult to recognize what is domain logic and what are other forms of logic. An informal test I like is to imagine adding a radically different layer to an application, such as a command-line interface to a Web application. If there's any functionality you have to duplicate in order to do this, that's a sign of where domain logic has leaked into the presentation. Similarly, do you have to duplicate logic to replace a relational database with an XML file?

A good example of this is a system I was told about that contained a list of products in which all the products that sold over 10% more than they did the previous month were coloured in red. To do this the developers placed logic in the presentation layer that compared this month's sales to last month's sales and if the difference was more than 10%, they set the colour to red.

The trouble is that that's putting domain logic into the presentation. To properly separate the layers you need a method in the domain layer to indicate if a product has improving sales. This method does the comparison between the two months and returns a Boolean value. The presentation layer then simply calls this Boolean method and, if true, highlights the product in red. That way the process is broken into its two parts: determining **if** there is something to highlight and choosing **how** to highlight.

I'm uneasy with being overly dogmatic about this. When reviewing this book, Alan Knight commented that he was "torn between whether just putting that into the UI is the first step on a slippery slope to hell or a perfectly reasonable thing to do that only a dogmatic purist would object to." The reason we are uneasy is because it's both!

## Choosing Where to Run Your Layers

Separation between layers is useful even if the layers are all running on one physical machine. However, there are places where the physical structure of a system makes a difference.

For most IS applications the decision is whether to run processing on a client, on a desktop machine, or on a server.

Often the simplest case is to run everything on servers. An HTML front end that uses a Web browser is a good way to do this. The great advantage of running on the server is that everything is easy to upgrade and fix because it's in a limited amount of places.

The general argument in favour of running on a client turns on responsiveness or disconnected operation. Any logic that runs on the server needs a server roundtrip to respond to anything the user does. If the user wants to fiddle with things and see immediate feedback, that roundtrip gets in the way. It also needs a network connection to run.

With those general forces in place, we can look at the options layer by layer. The data source pretty much always runs only on servers. The exception is where you might duplicate server functionality onto a suitably powerful client, usually when you want disconnected operation. In this case changes to the data source on the disconnected client need to be synchronized with the server.

The decision of where to run the presentation depends mostly on what kind of user interface you want. Running a rich client pretty much means running the presentation on the client. Running a Web interface pretty much means running on the server.

If you're building a B2C system, you have no choice. Anyone can be connecting to your servers and you don't want to turn anyone away. In this case you do all processing on the server and offer up HTML for the browser to deal with. Your limitation with the HTML option is that every bit of decision making needs a roundtrip from the client to the server, and that can hurt responsiveness. You can reduce some of the lag with browser scripting and downloadable applets, but they reduce your browser compatibility and tend to add other headaches. The more pure HTML you can go, the easier life is.

This leaves us with the domain logic. You can run business logic all on the server or all on the client, or you can split it. Again, all on the server is the best choice for ease of maintenance. The demand to move it to the client is for either responsiveness or disconnected use. The problem with putting all the domain logic on the client is that you have more to upgrade and maintain. Splitting across both the desktop and the server sounds like the worst of both worlds because you don't know where any piece of logic may be.

Once you've chosen your processing nodes, you should try to keep all the code in a single process. Don't try to separate the layers into discrete processes unless you absolutely have to. Doing that will both degrade performance and add complexity.

It's important to remember that many of these things are complexity boosters; distribution, explicit multithreading, multiplatform development, and extreme performance requirements (such as more than 100 transactions per second). All of these carry a high cost. Certainly there are times when you have to do it, but **never forget that each one carries a charge, both in development and in on-going maintenance.**

# Organizing Domain Logic

In organizing domain logic I've separated it into three primary patterns: **Transaction Script**, **Domain Model**, and **Table Module**.

The simplest approach to storing domain logic is the **Transaction Script**. A Transaction Script is essentially a procedure that takes the input from the presentation, processes it with validations and calculations, stores data in the database, and invokes any operations from other systems. It then replies with more data to the presentation, perhaps doing more calculation to help organize and format the reply.

The fundamental organization is of a single procedure for each action that a user might want to do. Hence, we can think of this pattern as being a script for an action, or business transaction. It doesn't have to be a single inline procedure of code. Pieces get separated into subroutines, and these subroutines can be shared between different Transaction Scripts. However, the driving force is still that of a procedure for each action, so a retailing system might have Transaction Scripts for checkout, for adding something to the shopping cart, for displaying delivery status, and so on.

A Transaction Script offers several advantages:

* It's a simple procedural model that most developers understand.
* It works well with a simple data source layer.
* It's obvious how to set the transaction boundaries: Start with opening a transaction and end with closing it.

Sadly, there are also plenty of disadvantages, which tend to appear as the complexity of the domain logic increases. Often there will be duplicated code as several transactions need to do similar things. Some of this can be dealt with by factoring out common subroutines, but even so much of the duplication is tricky to remove and harder to spot. The resulting application can end up being quite a tangled web of routines without a clear structure.

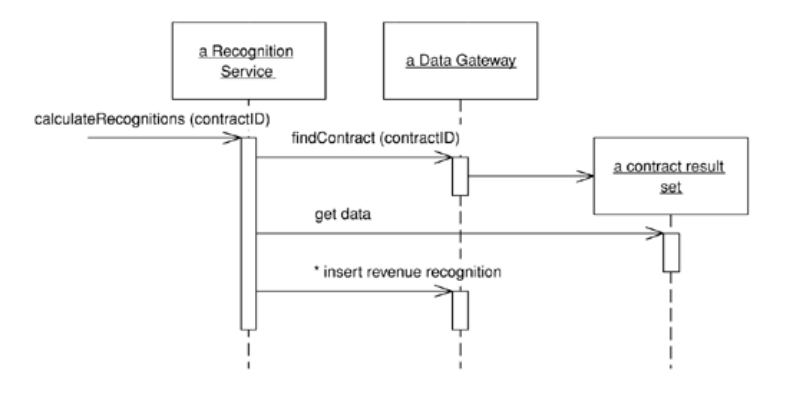


Figure 1 - A Transaction Scripts way of calculating revenue recognitions.

Of course, complex logic is where objects come in, and the object-oriented way to handle this problem is with a **Domain Model**. With a Domain Model we build a model of our domain which, at least on a first approximation, is organized primarily around the nouns in the domain. Thus, a leasing system would have classes for lease, asset, and so forth. The logic for handling validations and calculations would be placed into this domain model, so shipment object might contain the logic to calculate the shipping charge for a delivery. There might still be routines for calculating a bill, but such a procedure would quickly delegate to a Domain Model method.

Using a Domain Model as opposed to a Transaction Script is the essence of the paradigm shift that object-oriented people talk about so much. Rather than one routine having all the logic for a user action, each object takes a part of the logic that's relevant to it. If you're not used to a Domain Model, learning to work with one can be very frustrating as you rush from object to object trying to find where the behaviour is.

The value of a Domain Model lies in the fact that once you've gotten used to things, there are many techniques that allow you to handle increasingly complex logic in a well-organized way. With Transaction Script we're adding more conditions to the conditional logic of the script. Once your mind is as warped to objects as mine is, you'll find you prefer a Domain Model even in fairly simple cases.

The costs of a Domain come from the complexity of using it and the complexity of your data source layer. It takes time for people new to rich object models to get used to a rich Domain Model. Often developers may need to spend several months working on a project that uses this pattern before their paradigms are shifted. However, when you're used to Domain Model you're usually infected for life and it becomes easy to work with in the future.

Even once you've made the shift, you still have to deal with the database mapping. The richer your Domain Model, the more complex your mapping to a relational database. A sophisticated data source layer is much like a fixed cost; it takes a fair amount of money or time to get a good one, but once you have it you can do a lot with it.

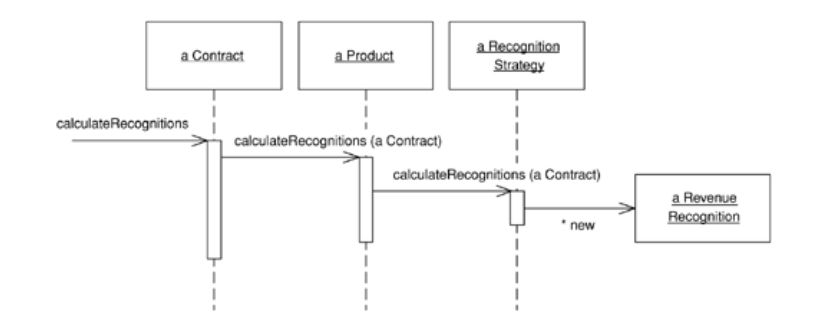


Figure 2 - A Domain Model's way of calculating revenue recognitions.

There's a third choice for structuring domain logic, **Table Module**. At very first blush the Table Module looks like a Domain Model, the vital difference is that a Domain Model has one instance of a class for each row in the database whereas a Table Module has only one instance for the entire table. A Table Module is designed to work with a Record Set. Thus, the client of a Table Module will first issue queries to the database to form a Record Set and will create a Table Model object and pass it the Record Set as an argument. The client can then invoke operations on the Table Module to do various things. If it wants to do something to an individual contract, it must pass in an ID.

A Table Module is in many ways a middle ground between a Transaction Script and a Domain Model. Organizing the domain logic around tables rather than straight procedures provides more structure and makes it easier to find and remove duplication. However, you can't use a number of the techniques that a Domain Model uses for finer grained structure of the logic, such as inheritance, strategies, and other OO patterns.

The biggest advantage of a Table Module is how it fits into the rest of the architecture. Many GUI environments are built to work on the results of a SQL query organized in a Record Set. Since a Table Module also works on a Record Set, you can easily run a query, manipulate the results in the Table Module, and pass the manipulated data to the GUI for display. You can also use the Table Module on the way back for further validations and calculations. A number of platforms, particularly Microsoft's COM and .NET, use this style of development.

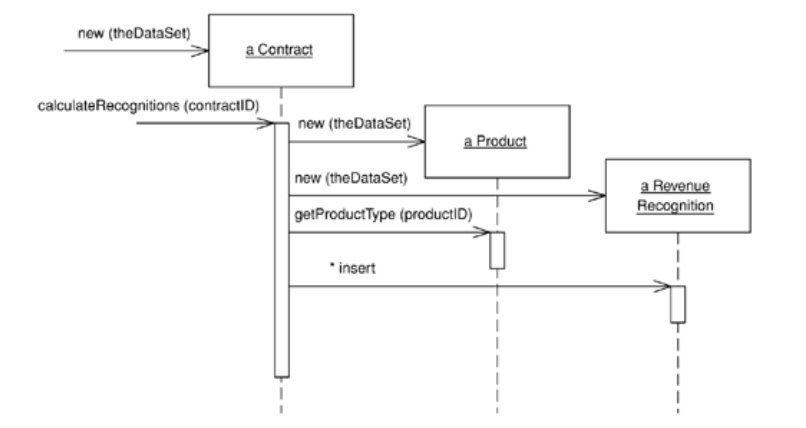
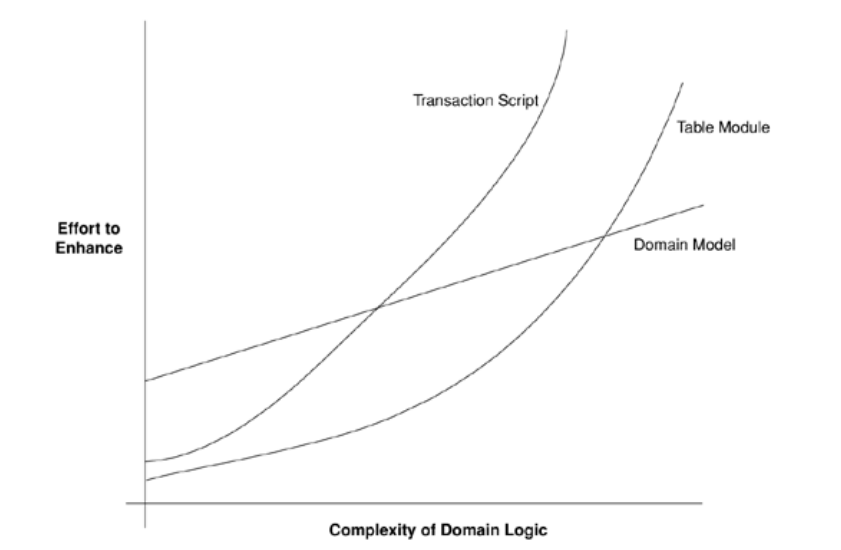


Figure 3- Calculating revenue recognitions with a Table Module

## Making a Choice

How do you choose between the three patterns? It's not an easy choice, and it very much depends on how complex your domain logic is. Figure 2.4 is one of those non-scientific graphs that really irritate me in PowerPoint presentations because they have utterly unquantified axes. However, it helps to visualize my sense of how the three compare. With simple domain logic the Domain Model is less attractive because the cost of understanding it and the complexity of the data source add a lot of effort to developing it that won't be paid back. Nevertheless, as the complexity of the domain logic increases, the other approaches tend to hit a wall where adding more features becomes exponentially more difficult.



Your problem, of course, is to figure out where on that x axis your application lies. In practice all you can do is find some experienced people who can do an initial analysis of the requirements and make a judgment call.

There are some factors that alter the curves a bit. A team that's familiar with Domain Model will lower the initial cost of using this pattern. It won't lower it to same starting point as the others because of the data source complexity. Still, the better the team is, the more I'm inclined to use a Domain Model.

The attractiveness of a Table depends very much on the support for a common Record Set structure in your environment. If you have an environment like .NET or Visual Studio, where lots of tools work around a Record Set, then that makes a Table Module much more attractive. Indeed, I don't see a reason to ever use Transaction Scripts in a .NET environment.

## Service Layer

A common approach in handling domain logic is to split the domain layer in two. A Service Layer *(also referred to as the Application Service)* is placed over an underlying Domain Model or Table Module. The presentation logic interacts with the domain purely through the Service Layer which acts as an API for the application.

As well as providing a clear API, the Service Layer is also a good spot to place such things as transaction control and security. This gives you a simple model of taking each method in the Service Layer and describing its transactional and security characteristics. A separate properties file is a common choice for this, but .NET's attributes provide a nice way of doing it directly in the code.

When you see a Service Layer, a key decision is how much behaviour to put in it. The minimal case is to make the Service Layer a facade so that all of the real behaviour is in underlying objects and all the Service Layer does is forward calls on the facade to lower-level objects. In that case the Service Layer provides an API that's easier to use because it's typically oriented around use cases. It also makes a convenient point for adding transactional wrappers and security checks. At the other extreme, most business logic is placed in Transaction Scripts inside the Service Layer and the underlying domain objects are very simple.

# Mapping to Relational Databases

The role of the data source layer is to communicate with the various pieces of infrastructure that an application needs to do its job. A dominant part of this problem is talking to a database, which, for the majority of systems built today, means a relational database.

## Architectural Patterns

The choice you make here is far-reaching for your design and thus difficult to refactor, so it's one that you should pay some attention to. It's also a choice that's strongly affected by how you design your domain logic.

### Table Data Gateway

Mixing SQL in application logic can cause several problems. Many developers aren't comfortable with SQL, and many who are comfortable may not write it well. Database administrators need to be able to find SQL easily so they can figure out how to tune and evolve the database.

A Table Data Gateway holds all the SQL for accessing a single table or view: selects, inserts, updates, and deletes. Other code calls its methods for all interaction with the database.

A Table Data Gateway has a simple interface, usually consisting of several find methods to get data from the database and update, insert, and delete methods. Each method maps the input parameters into a SQL call and executes the SQL against a database connection. The Table Data Gateway is usually stateless, as its role is to push data back and forth.

One of the key uses for a Table Data Gateway is as a good point at which to apply a Service Stub in order to remove any dependency on the database during unit testing.

Table Data Gateway fits very nicely with Table Module and Transaction Script.

### Data Mapper

Objects and relational databases have different mechanisms for structuring data. Many parts of an object, such as collections and inheritance, aren't present in relational databases. When you build an object model with a lot of business logic it's valuable to use these mechanisms to better organize the data and the behaviour that goes with it. Doing so leads to an object schema and relational schema that don't match up.

You still need to transfer data between the two schemas, and this data transfer becomes a complexity in its own right. If the in-memory objects know about the relational database structure, changes in one tend to ripple to the other.

The Data Mapper is a layer of software that separates the in-memory objects from the database. Its responsibility is to transfer data between the two and also to isolate them from each other. With Data Mapper the in-memory objects needn't know even that there's a database present; they need no SQL interface code, and certainly no knowledge of the database schema.

Data Mapper fits nicely with a Domain Model from the database completely as it is responsible for the mapping between domain objects and database tables. The Data Mapper handles all of the loading and storing between the database and the Domain Model and allows both to vary independently. It's the most complicated of the database mapping architectures, but its benefit is complete isolation of the two layers.

### Repository

A Repository mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.

A system with a complex domain model often benefits from a layer, such as the one provided by Data Mapper, that isolates domain objects from details of the database access code. In such systems it can be worthwhile to build another layer of abstraction over the mapping layer where query construction code is concentrated. This becomes more important when there are a large number of domain classes or heavy querying. In these cases particularly, adding this layer helps minimize duplicate query logic.

A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection. Client objects construct query specifications declaratively and submit them to Repository for satisfaction. Objects can be added to and removed from the Repository, as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes. Conceptually, a Repository encapsulates the set of objects persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer. Repository also supports the objective of achieving a clean separation and one-way dependency between the domain and data mapping layers.

### Query Object

SQL can be an involved language, and many developers aren't particularly familiar with it. Furthermore, you need to know what the database schema looks like to form queries. You can avoid this by creating specialized finder methods that hide the SQL inside parameterized methods, but that makes it difficult to form more ad hoc queries.

A particularly sophisticated use of Query Object is to eliminate redundant queries against a database. If you see that you've run the same query earlier in a session, you can use it to select objects from the cache and avoid a trip to the database.

### Object Relational Mapper

It is however worth considering the use of an ORM tool if you have a Domain Model. While the patterns in this book will tell you a lot about how to build a Data Mapper, it's still a complicated endeavour. Commercial ORM tools are much more sophisticated than anything that can reasonably be done by hand. Good ORM tools give you a lot of options in mapping to a database, and these patterns will help you understand when to use the different choices.

Don't assume that a tool makes all the effort go away. It makes a big dent, but you'll still find that using and tuning an ORM takes a small but significant chunk of work.

### Active Record

### Memento

## The Behavioural Problem

Training note: This section is provided as way of familiarising yourself with these concepts. Modern ORM tools such as Entity Framework take care of all the concerns mentioned below for you in a relatively transparent manner,

When people talk about O/R mapping, they usually focus on the structural aspects; how you relate tables to objects. However, I've found that the hardest part of the exercise is its architectural and behavioural aspects.

That behavioural problem is how to get the various objects to load and save themselves to the database. If you load a bunch of objects into memory and modify them, you have to keep track of which ones you've modified and make sure to write all of them back out to the database. If you only load a couple of records, this is easy. As you load more and more objects it gets to be more of an exercise, particularly when you create some rows and modify others since you'll need the keys from the created rows before you can modify the rows that refer to them.

As you read objects and modify them, you have to ensure that the database state you're working with stays consistent. If you read some objects, it's important to ensure that the reading is isolated so that no other process changes any of the objects you've read while you're working on them. Otherwise, you could have inconsistent and invalid data in your objects. This is the issue of concurrency, which is a very tricky problem to solve.

A pattern that's essential to solving both of these problems is Unit of Work. A Unit of Work keeps track of all objects read from the database, together with all objects modified in any way. It also handles how updates are made to the database. Instead of the application programmer invoking explicit save methods, the programmer tells the unit of work to commit. That unit of work then sequences all of the appropriate behaviour to the database, putting the entire complex commit processing in one place. Unit of Work is an essential pattern whenever the behavioural interactions with the database become awkward.

As you load objects, you have to be wary about loading the same one twice. If you do that, you'll have two in-memory objects that correspond to a single database row. Update them both, and everything gets very confusing. To deal with this you need to keep a record of every row you read. Each time you need read in some data you have check the existing data to make sure that you don't already have it. If the data is already loaded, you can return a second reference to it. That way any updates will be properly coordinated.

If you're using a Domain Model, you'll usually arrange things so that linked objects are loaded together in such a way that a read for an order object loads its associated customer object. However, with many objects connected together any read of any object can pull an enormous object graph out of the database. To avoid such inefficiencies you need to reduce what you bring back yet still keep the door open to pull back more data if you need it later on. Lazy Loading relies on having a placeholder for a reference to an object. There are several variations on the theme, but all of them have the object reference modified so that, instead of pointing to the real object, it marks a placeholder. Only if you try to follow the link does the real object get pulled in from the database. Using Lazy Load at suitable points, you can bring back just enough from the database with each call.