**Part 1: - The Narratives**

**Chapter 1: Layering**

* 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. Furthermore, each layer usually hides its lower layers from the layers above, so layer 4 uses the services of layer 3, which uses the services of layer 2, but layer 4 is unaware of layer 2.
* Layering is one of the most common techniques that software designers use to break apart a complicated software system and it has several important benefits.

1. Understand single layer without knowing much about the other layers
2. You can substitute layers with alternative implementations of the same basic services
3. Once you have a layer built, you can use it for many higher-level services

* Layering is an important technique, but there are downsides.

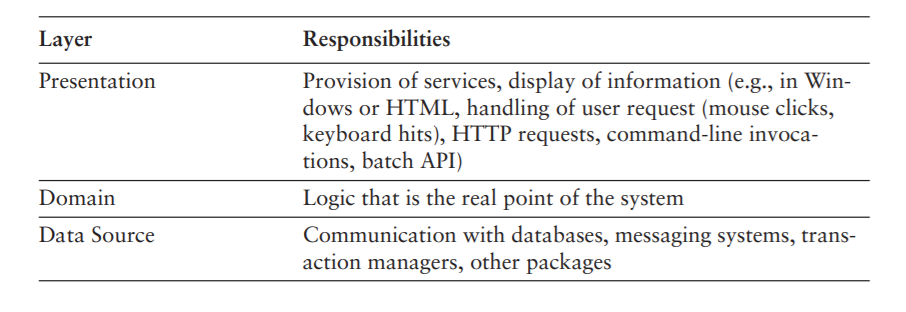
1. Layers encapsulate some, but not all, things well. As a result, you sometimes get cascading changes. The classic example of this in a layered enterprise application is adding a field that needs to display on the UI, must be in the database, and thus must be added to every layer in between.
2. Extra layers can harm performance. At every layer thing typically need to be transformed from one representation to another

* ***The evolution of layers in enterprise application***

1. 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.
2. 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
3. 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.
4. 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
5. While 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

* ***The Three Principal layer***

1. For this book the centering would be around an architecture of three primary layers: presentation, domain, and data source.
2. Presentation logic is about how to handle the interaction between the user and the software.
3. the presentation layer is to display information to the user and to interpret commands from the user into actions upon the domain and data source
4. domain logic, also referred to as business logic. This is 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
5. Data source logic is about communicating with other systems that carry out tasks on behalf of the application. These can be transaction monitors, other applications, messaging systems, and so forth. For most enterprise applications the biggest piece of data source logic is a database that is primarily responsible for storing persistent data

******

* ***Choosing where to run your layer***

1. For most of this book I will be talking about logical layers—that is, dividing a system into separate pieces to reduce the coupling between different parts of a system. 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.
2. 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
3. The great advantage of running on the server is that everything is easy to upgrade and fix because it’s in a limited number of places.
4. The general argument in favor 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
5. Disconnected operation brings particular challenges where users want to do work without internet connection
6. 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
7. 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.
8. 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.
9. If you have to run some logic on the client, you can consider running all of it there—at least that way it’s all in one place. Usually this goes hand in hand with a rich client—running a Web server on a client machine isn’t going to help responsiveness much, although it can be a way to deal with disconnected operation
10. The problem with putting all the domain logic on the client is that you have more to upgrade and maintain.
11. 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.
12. The main reason to do it is that you have only a small amount of domain logic that needs to run on the client , The trick then is to isolate this piece of logic in a self-contained module that isn’t dependent on any other part of the system.

**Chapter** **2: - Organizing Domain Logic**

1. In organizing domain logic I’ve separated it into three primary patterns: Transaction Script, Domain Model , and Table Module.
2. **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
3. 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
4. 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 using Row Data Gateway or Table Data Gateway .

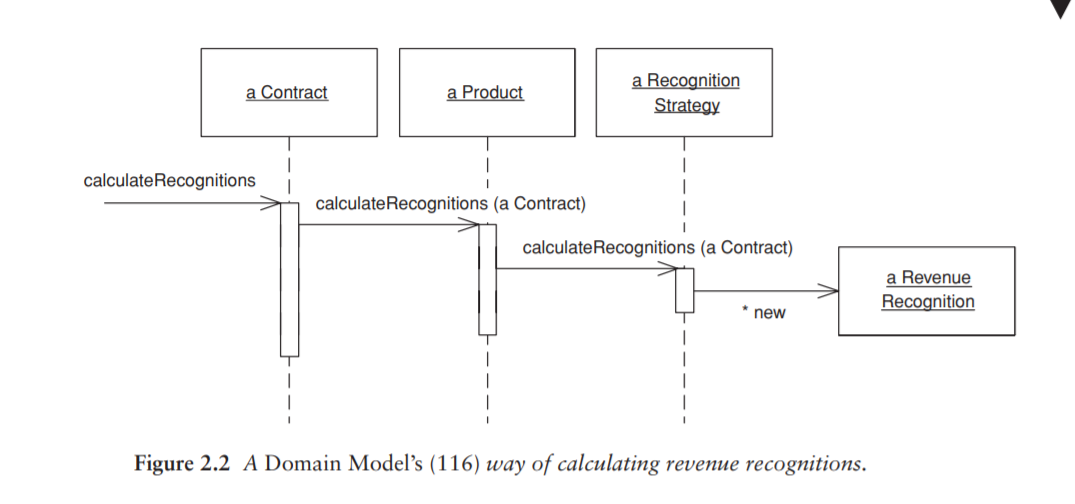
• It’s obvious how to set the transaction boundaries: Start with opening a transaction and end with closing it. It’s easy for tools to do this behind the scenes.

1. 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.
2. complex logic is where objects come in, and the object-oriented way to handle this problem is with a Domain Model
3. 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

Diagram

Description automatically generated

1. 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 learning to work with one can be very frustrating as you rush from object to object trying to find where the behavior is Domain Model.

****

1. There’s a third choice for structuring domain logic, Table Module . At very first blush the Table Module looks like a Domain Model since both have classes for contracts, products, and revenue recognitions. The vital difference is that a Domain Model has one instance of contract for each contract in the database whereas a Table Module has only one instance. A Table Module is designed to work with a Record Set .
2. The client of a contract **Table Module** will first issue queries to the database to form a Record Set (508) and will create a contract object and pass it the Record Set

Diagram

Description automatically generated

* ***Making a choice***

Diagram

Description automatically generated

* ***Service Layer***

1. A common approach in handling domain logic is to split the domain layer in two. A Service Layer is placed over an underlying Domain Model or Table Module.
2. As well as providing a clear API, the Service Layer (133) 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 (133) and describing its transactional and security characteristics.
3. When you see a Service Layer (133), a key decision is how much behavior to put in it. The minimal case is to make the Service Layer (133) a facade so that all of the real behavior is in underlying objects and all the Service Layer (133) does is forward calls on the facade to lower-level objects.
4. In that case the Service Layer (133) provides an API that’s easier to use because it’s typically oriented around use cases.
5. At the other extreme, most business logic is placed in Transaction Scripts inside the Service Layer (133). The underlying domain objects are very simple; if it’s a Domain Model it will be one-to-one with the database, and you can thus use a simpler data source layer such as Active Record
6. At the other extreme, most business logic is placed in Transaction Scripts (110) inside the Service Layer (133). The underlying domain objects are very simple; if it’s a Domain Model (116) it will be one-to-one with the database, and you can thus use a simpler data source layer such as Active Record
7. These are different controllers to the input controller in Model View Controller (330) or Application Controller (379) that we’ll meet later, so I use the term **use-case controller**. **Behavior that’s used in more than one use case goes on the domain objects**, which are called entities. Although the **controller-entity approach** is a common one, it’s not one that I’ve ever liked much. The use case controllers, like any Transaction Script (110), tend to encourage duplicate code. My view is that, if you decide to use a Domain Model (116) at all, you really should go whole hog and make it dominant. The one exception to this is if you’ve started with a design that uses Transaction Script (110) with Row Data Gateway (152). Then it makes sense to move duplicated behavior to the Row Data Gateways (152), which will turn them into a simple Domain Model (116) using Active Record (160). However, I wouldn’t start that way. I would only do that to improve a design that’s showing cracks.