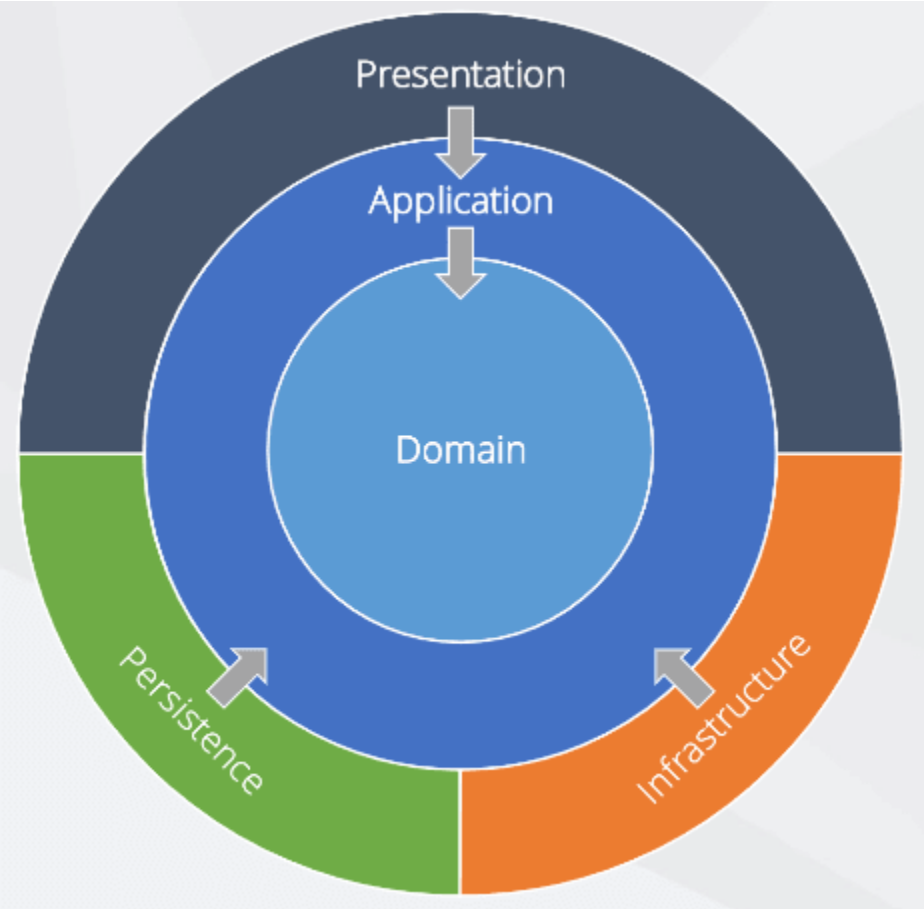
Clean Architecture

The logical conclusion of the Stable Abstractions and Stable Dependencies principles is Clean (AKA Onion) Architecture. There are many variations of the Clean Architecture Layer model some with as many as five layers and as few as three. I prefer the simple model because it is easier to conceptualize:



The arrows in this diagram indicate the direction of the dependencies. Each outer layer can depend on any of the layers beneath it but none of the inner layers can depend (directly) on any of the outer layers. The inner layers can, however, define interfaces that the outer layers can implement. In this manner the inner layers are dependent on the abstraction (interface) but not the concrete implementation (class).

So time for some definitions. The Domain layer is where the business rules go.

> *Business rules are structures, calculations, processes, or constraints that the organization adheres to as a function of its competitive advantage or of the regulatory requirements in the legal jurisdictions in which it operates.*

If we accept this as the definition of a business rule what we must then understand and accept is that they do not change very often. How often to laws change? The way interest is calculated? The types of entities and their relationship the system manages? The fundamental nature of what makes our business different from our competitors? So how often do these things change…seldom…almost never? Therefore, it is perfectly acceptable, and in-fact desirable to hard code these rules in the Domain layer where they can be unit tested and validated.

Some things that are not business rules:

- product categories
- pricing or promotional tiers
- rates (tax, interest, etc) unless is its a well known constant like the acceleration due to gravity
- product configurations
- Any optional or demographic field on an entity. (Think about it, if you are making business decisions based on someone's name you are very likely breaking numerous laws and regulations). You may have a structural rule that involves a Customer but in that case the Domain entity can expose a Customer ID and the application layer can map that to a View Model or DTO that contains the customer's name. **Only the fields required by the business rule should be defined on the Domain entity.**

Along with business rules, the Domain layer should also define business events and track which business events should be raised in response to changes to the entities it defines. Note, I did not say the Domain layer should actually raise these events because raising events, by their nature, results in side effects and out of process calls which can impact performance as well as have transaction scope implications. The Application layer, which I will cover next, understands the transaction context and the performance implications of various calls and is therefore responsible for actually raising the Domain event. Finally the Domain layer should be responsible for state. By that I do not mean directing the persistence of state, that again should be the responsibility of the Application layer, what I mean is the Domain layer should be responsible for asserting which

operations are allowed and which are not under a set of predefined conditions. This is best accomplished using the GOF State Pattern. The unfortunate definition of this pattern is often given as: “allows an object to behave differently depending on its internal state” which is ambiguous as to when and where you would apply it, however, when you look at how it is actually implemented you arrive at an improved definition of the State Pattern “*asserting which operations are allowed and which are not under a set of predefined conditions*”.

The Application layer is where you implement the orchestration between inputs and outputs as well as the management of the transaction scope. The Application layer is where you raise Domain events, invoke repository (persistence and cache) interfaces, commit transactions, raise validation, argument, and concurrency exceptions, handle transient exceptions, and implement the sequence of required operations to realize the applications behavior. The Application layer is responsible for consulting the Domain layer but unit tests are required to insure it actually does. As a natural consequence of its location in the onion the Application layer is responsible for mapping View Models, DTOs, and Entities but should not have any direct IO responsibilities. There should not be any references to HttpContext, HttpClient, HttpClientFactory, Dapper, or Sql.Data. Anything, the application layer should be completely unit testable using F.I.R.S.T principles on a laptop that does not have a network connection.

The very outer layer in the model (persistence, presentation, and infrastructure) is the I/O layer. The classes in this layer must implement the interfaces defined in the lower layers. Being I/O bound, these are Humble Objects and are not testable using F.I.R.S.T principles. they can, however be the subject of system tests. Since these objects cannot be unit tested, we **MUST NEVER** implement business rules inside these classes.

But…wait, you say: What if I have a business rule that results in a constraint on the type of data we return from the database, for performance reasons, it would be better to enforce this constraint in the WHERE clause. But ahhh, I reply Specification Pattern to the rescue. There is always a better way. If you ever feel you *MUST* implement a business rule in the I/O layers or repositories, come see me and well find a better way together. At this point I can here the API developers breathing a sign of relief because they do not have a “presentation” layer…ahh…but you do. You do not have a U/I layer but your API controller is the presentation layer of your service.