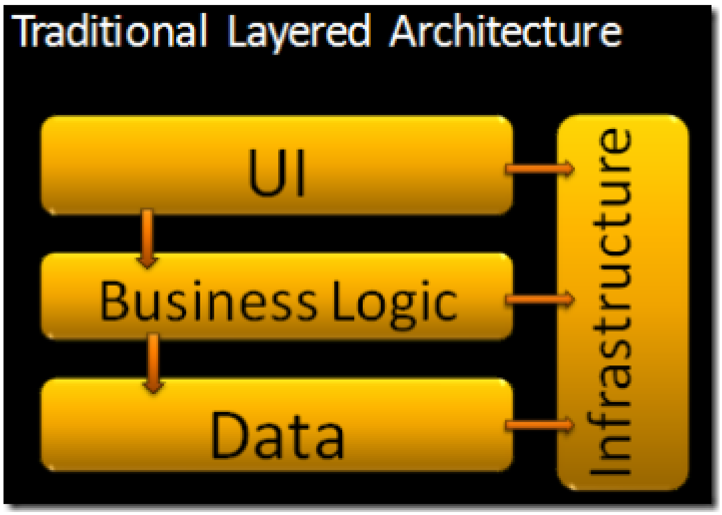
Jeffrey Palermo 8:08 am on July 29, 2008

Tags: onion architecture (4)

The Onion Architecture : part 1

This is part 1. part 2. part 3. part 4. My feed (rss).

I've spoken several times about a specific type of architecture I call "Onion Architecture". I've found that it leads to more maintainable applications since it emphasizes separation of concerns throughout the system. I must set the context for the use of this architecture before proceeding. This architecture is not appropriate for small websites. It is appropriate for long-lived business applications as well as applications with complex behavior. It emphasizes the use of interfaces for behavior contracts, and it forces the externalization of infrastructure. The diagram you see here is a representation of traditional layered architecture. This is the basic architecture I see most frequently used. Each subsequent layer depends on the layers beneath it, and then every layer normally will depend on some common infrastructure and utility services. The big drawback to this top-down layered architecture is the coupling that it creates. Each layer is coupled to the layers below it, and each layer is often coupled to various infrastructure concerns. However, without coupling, our systems wouldn't do anything useful, but this architecture creates unnecessary coupling.

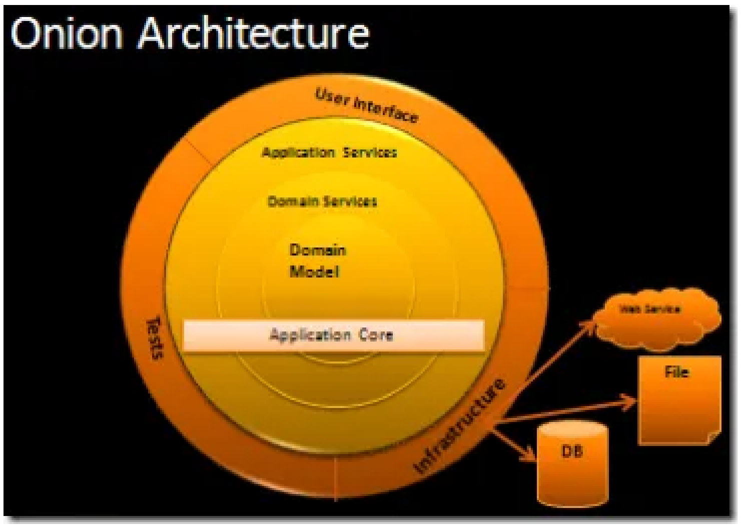


The biggest offender (and most common) is the coupling of UI and business logic to data access. Yes, UI is coupled to data access with this approach. Transitive dependencies are still dependencies. The UI can't function if business logic isn't there. Business logic can't function if data access isn't there. I'm intentionally ignoring infrastructure here because this typically varies from system to system. Data access changes frequently. Historically, the industry has modified data access techniques at least every three years; therefore, we can count on needing to modify data access three years from now for any healthy, long-lived systems that's mission-critical to the business. We often don't keep systems up-to-date because it's impossible to do. If coupling prevents easily upgrading parts of the system, then the business has no choice but to let the system fall behind into a state of disrepair. This is how legacy systems become stale, and eventually they are rewritten.

I propose a new approach to architecture. Honestly, it's not completely new, but I'm proposing it as a named, architectural pattern. Patterns are useful because it gives software professionals a common vocabulary with which to communicate. There are a lot of aspects to the Onion Architecture, and if we have a common term to describe this approach, we can communicate more effectively.

The diagram to the left depicts the Onion Architecture. The main premise is that it controls coupling. The fundamental rule is that all code can depend on layers more central, but code cannot depend on layers further out from the core. In other words, all coupling is toward the center. This architecture is unashamedly biased toward object-oriented programming, and it puts objects before all others.

In the very center we see the Domain Model, which represents the state and behavior combination that models truth for the organization. Around the Domain Model are other layers with more behavior. The number of layers in the application core will vary, but remember that the Domain Model is the very center, and since all coupling is toward the center, the Domain Model is only coupled to itself. The first layer around the Domain Model is typically where we would find interfaces that provide object saving and retrieving behavior, called repository interfaces. The object saving behavior is not in the application core, however, because it typically involves a database. Only the interface is in the application core. Out on the edges we see UI,



Infrastructure, and Tests. The outer layer is reserved for things that change often. These things should be intentionally isolated from the application core. Out on the edge, we would find a class that implements a repository interface. This class is coupled to a particular method of data access, and that is why it resides outside the application core. This class implements the repository interface and is thereby coupled to it.

The Onion Architecture relies heavily on the [Dependency Inversion principle](). The application core needs implementation of core interfaces, and if those implementing classes reside at the edges of the application, we need some mechanism for injecting that code at runtime so the application can do something useful.

The database is not the center. It is external. Externalizing the database can be quite a change for some people used to thinking about applications as “database applications”. With Onion Architecture, there are no database applications. There are applications that might use a database as a storage service but only though some external infrastructure code that implements an interface which makes sense to the application core. Decoupling the application from the database, file system, etc, lowers the cost of maintenance for the life of the application.

[Alistair Cockburn]() has written a bit about [Hexagonal architecture](). Hexagonal architecture and Onion Architecture share the following premise: Externalize infrastructure and write adapter code so that the infrastructure does not become tightly coupled.

I’ll be writing more about the Onion Architecture as a default approach for building enterprise applications. I will stay in the enterprise system space and all discussion will reside in that context. This gets even more interesting when there are multiple processes making up a single software system.