Objetivo: recap OOP-Architecture Sesión II
Autor: Marcos Chotsourian
Fecha: 20/01/12

Recap OOP-Architecture Sesión II

En la segunda sesión de OOP-Architecture continuamos imaginando cómo sería una aplicación bancaria, pensando funciones específicas que debe realizar la aplicación. En líneas generales, completamos el esquema de la arquitectura y continuamos pensando cómo y cuándo utilizar clases, clases abstractas e interfaces.

--Esquema de arquitectura API propuesto:

Proyecto API:

Controller: Inicia solicitudes.

IServices: Declara los roles que tramitan las solicitudes.

Services: Declara las clases que implementan esos roles.

Injección de dependencias en clase Program.cs

Proyecto Logic:

ILogic: Declara los roles que interactúan con la base de datos.

Logic: Declarar las clases que implementan esos roles.

Proyecto Data:

ServiceContext.cs : Define la estructura de la base de datos por EF.

Proyecto Entities:

Clases que se utilizan a lo largo de toda la aplicación. Ya sea para mappear tablas en la base de datos o no.

Para pensar y darle algunas vueltas…

¿Qué es la injección de dependencias y por qué se usa?

¿Por qué la lógica y los servicios se trabajan con interfaces?

¿Por qué intermediamos entre el Proyecto API y el Proyecto Data con el de Logic?

¿Qué situaciones de trabajo real en la industria del software son mejores o peores según lo mejor o peor que hayamos hecho la arquitectura?

Objetivo: recap OOP-Architecture Sesión II
Autor: Marcos Chotsourian
Fecha: 20/01/12

-- Cómo y cuándo utilizar clases, clases abstractas e interfaces

Clase abstracta: la utilizamos cuando necesitamos definir clases hijas que hereden tanto atributos y métodos definidos **como métodos abstractos**. Esta clase no se puede instanciar.

Clase: la utilizamos cuando necesitamos definir clases hijas que hereden atributos y **solo métodos definidos**. Esta clase se puede instanciar.

Por ejemplos:

Podría ser que en mi aplicación yo tenga la clase abstracta Cobro, que tenga como **atributos** Valor, Fecha, ModoDePago, etcs y como **métodos abstractos** Revertir, DeducirImpuestos, etcs. Esta clase abstracta puede tener como hijos a CobroEnEfectivo y CobroConTarjeta. Esto sería indicado en el caso de que en mi aplicación no se necesite trabajar puntualmente con instancias de Cobro en general, sino con instancias especificadas de Cobro.

También podría ser que en mi aplicación yo tenga la clase **no** abstracta Cobro, que tenga los mismos atributos y métodos de antes (pero definidos) y los mismos hijos. La diferencia es que en este caso la aplicación sí necesita trabajar con instancias de Cobro puntuales, sin importar qué tipo de Cobro son.

Las interfaces, recordemos, definen un rol-del-código que puede tomar una clase. Todas las clases que implementan una interfaz están forzadas a declarar ciertos métodos y por ello es que pueden tomar ese rol.

Las interfaces las usaremos para definir dependencias (que son roles de servicios de la API o de lógicas de Logic). Esto sirve para que la aplicación no dependa de tal o cual forma de brindar un servicio ni de tal o cual forma de interactuar con una base de datos.

También podemos utilizarlas para nombrar varias clases que cumplan un rol pero que no sean del mismo tipo de clase. Por ejemplo, IGravable puede definir un rol que cumplan todas las clases que son gravables impositivamente. Esto se haría con una interfaz y no con una clase padre porque cumplir ese rol no implica que sean del mismo tipo ni que necesiten heredar todas los mismos atributos o métodos.

La abstracción que logramos con una clase padre es generalidad.

La abstracción que logramos con una interfaz es un rol.

Para pensar y darle algunas vueltas…

¿Siempre que podamos debemos trabajar con toda la abstracción posible? ¿Cuándo sí y cuándo no?

¿En qué ejemplos concretos de aplicación de negocio se les ocurre que se puede utilizar una clase padre, una clase abstracta padre y una interfaz?

Objetivo: recap OOP-Architecture Sesión II
Autor: Marcos Chotsourian
Fecha: 20/01/12
--Para terminar por ahora…

OOP y Architectura son cuestiones difíciles que toman tiempo. A medida que trabajen en aplicaciones reales y grandes van a seguir descubriendo cuestiones conceptuales sobre ello y notando la gran importancia que tienen para el negocio. Por más que una aplicación pueda "funcionar" con una mala arquitectura, es muy importante tratar de tenerlo presente porque facilita mucho el trabajo de desarrollo y manutención del código así como el poder trabajar colaborativamente con otros equipos o con aplicaciones creadas hace muchos años.

No tomen esto como una serie de reglas fijas a seguir, sino que llévense criterios de pensamiento a partir de los cuales pueden ir teniendo en cuenta estas cuestiones. Priorizando que la aplicación funcione, cada cierto tiempo deténganse en la arquitectura para tratar de darle una vuelta y ver si es posible hacerla de una mejor manera.

Tampoco tienen por qué hacerla tal cual el ejemplo brindado, si ven una manera de hacerla mejor, ¡adelante!

Las invito a responder y trabajar en esta recap y compartir lo logrado.