RETO 2 – Solución del Profesor

¿Qué tengo que hacer qué?

Lo que tienes ante ti es el **Reto 2 de la Agencia de Viajes** hecho por el profe siguiendo un montón de criterios de programación avanzadas. Es decir, que me he molestado en hacer el Reto 2 forzando un montón de cosas para que tengas una visión básica de cómo se ve un proyecto real en la empresa. Insisto en que sólo es una aproximación a cómo se trabaja en proyectos reales.

En este proyecto, se usan cosas que se van a ver en la tercera evaluación que puedes encontrar en el proyecto. Por ejemplo:

- Herencia
- Polimorfismo
- Sobrecarga
- Abstracción
- Interfaces
- Excepciones propagadas, nuevas excepciones

A demás, incluyo cosas que no entran estrictamente hablando, pero que nos tienen que sonar:

- Patrón Singleton
- Patrón Factory
- Fichero de Propiedades

Y una cosa de la que os han hablado, pero no entendíais bien:

- Vista – Controlador – Modelo (MVC)

Antes de que digas nada, instala el proyecto y míratelo. Fíjate en que:

- No está hecho entero. Sólo una parte.
- Los métodos de acceso a BBDD son de pruebas, no usan la BBDD.

El objetivo de este ejercicio es: Añade el panel de Registro del Agencia, pero manteniendo la estructura del proyecto. No hace falta que añadas lo de los colores corporativos ni lo de la imagen.

Have fun!

Os explico algo de qué va esto…

Primero. MCV es un patrón de diseño. Es una manera de programar proyectos de manera que coloques las cosas por **capas**. En este caso:

- **Vista**: Las clases de la vista están en el paquete **view**. En la vista de una aplicación únicamente deberían de estar las cosas que tengan que ver con recoger información y mostrar información. Nunca deberían de 'tomar decisiones', de ningún tipo.

- **Controlador**: Las clases del controlador toman decisiones. Están en el paquete **service**. Deciden lo que hay que hacer cada vez que alguien pulsa un botón en una ventana. Si hay que cargar una tabla, y para eso hay que hacer tres consultas a la BBDD, se encargan ellos. Pero ojo: no saben cómo funciona la Base de Datos ni saben cómo cargar la tabla.

- **Modelo**: Están en el paquete **dataBase**. Estas clases se encargan de acceder al modelo de datos, que en el 99'99% de las veces es una Base de Datos de alguna clase.

Por tanto, para programar una aplicación como es debido según el patrón MCV, hay que separar bien el código es estas tres partes. Y más cosas.

P: ¿Es posible ver en un proyecto las tres capas MCV separadas en paquetes así?

R: Lo normal es que se use MCV y todo lo que cuento aquí, pero normalmente no hay un paquete 'vista' con todo lo de las ventanitas.

P: ¿Cómo sé qué clase meto en cada lugar?

R: Sentido común. Escribir un fichero de texto no tiene mucho sentido meterlo en la parte de la vista o del modelo, ¿verdad? Esto lo deciden los Analistas, gente que diseña el programa.

P: ¿Tengo que pasar siempre por el Controlador para llegar al Modelo?

R: Si. No puedes saltarte las capas. Que de la vista llames a una función de la Base de Datos es un error. Te cargas el patrón MCV. Aunque el método al que llames en el controlador no haga nada más que llamar a otro en el modelo.

P: ¿Para qué tanta complicación? ¿No es más fácil…?

R: NO, NO LO ES. Las ventajas de hacer un MCV las explico más adelante, pero se resume a una bien simple: trabajas ahora para facilitarte la vida en el futuro. Como, por ejemplo, cuando tengas que arreglar algo o añadir algo a un proyecto que lleva 7 años funcionando. Si se ha seguido MCV, tu trabajo será simple y no requerirás aspirinas.

Cómo está el programa montado

Pues mira, te cuento…

Aunque no se usen para nada, he pensado que sería entretenido añadir la configuración de la base de datos en el fichero de texto **config.properties**. Este fichero se lee automáticamente al arrancar, de forma que, si cambiamos la BBDD, basta con cambiar este fichero. Esto se hace mucho con cosas así. Puedes ver cómo lo hago en la clase **DataBaseConfig**. Esta clase tiene un Singleton.

Cada una de las tres capas tiene un Factory y un Singleton. Esto significa dos cosas:

1) Las clases **Factory** siempre estarán en memoria porque son también **Singleton**. No hay que hacer un new cada vez que queramos usarlas, y siempre habrá al menos una copia en memoria de ese Factory.

2) Las clases **Factory** centralizan la creación de **Panels**, **Controllers** y **Managers**. Dicho de otro modo, le pides al Factory que te pase el Panel, o el Controller o el Manager que quieres usar.

   Esto tiene varias ventajas. Para empezar, esto desacopla las capas. Esto quiere decir que las clases de una capa no dependen de las de otra capa. Dicho de otro modo, que a las clases de la Vista les da igual cómo son las clases del Controlador, y a estas, las del Modelo.

   O sea, mira este ejemplo del **LoginPanelController**. Necesita usar **AgencyManager** para usar el método **getAgency(agency, pass)**, pero en lugar de hacer new se lo pide al **DatabaseFactory** y lo usa.

```
private Agency getAgency(String agency, String pass) {
    AgencyManager agencyManager = (AgencyManager) DatabaseFactory.getInstance()
            .getManager(DatabaseFactory.panelOptions.AGENCY_MANAGER.value);
    return agencyManager.getAgency(agency, pass);
}
```

   El método NO necesita conocer cómo es el AgencyManager. Simplemente se lo pide al Factory, y usa el método. Y pista. Si no usase un **Factory**, el código sería:

```
private Agency getAgencyNotFactory(String agency, String pass) {
    AgencyManager agencyManager = new AgencyManager();
    return agencyManager.getAgency(agency, pass);
}
```

   El problema es que hacer esto obliga al **LoginPanelController** a conocer el **AgencyManager.** Está acoplado a una clase de otra capa. Y lo que es peor… suponte que ahora hay que cambiar el **AgencyManager** por un **EnterpriseManager** en siete sitios. Menudo jaleo tener que ir por todo el código arreglando movidas. En cambio, al usar el Factory… te da igual. Añades un nuevo **EnterpriseManager** al Factory y quien lo quiera, que lo pida. Fácil.

A demás, al desacoplar capas lo que conseguimos es que varias personas puedan trabajar a la vez en el mismo proyecto. Uno se encarga de los Panels, otro hace los Controllers, y otro los Managers. Si están de acuerdo en cómo se van a llamar las clases y los métodos, entonces no se chocan entre ellos.

Es un poco difícil de verlo. No te líes. Muchas de estas cosas son de Diseño de Aplicaciones, lo que hace un Analista antes de empezar a programar.