Trabajo Práctico Grupal
“Sistema de Gestión de una Clínica”
Primera Etapa

Año: 2021
Fecha de entrega: 16/5/2021

Integrantes: Fioriti, Joaquín
García, Mariano
Pruis, Ramiro
Sardi, Franco

Patrones de diseño implementados

Patron Singleton

Este patrón es utilizado cuando solo necesitaremos una instancia de determinada clase. De esta manera, evitamos el tener que pasar por parámetro la clase y podremos acceder a la instancia desde cualquier parte del programa que importe dicha clase.

La dinámica es la siguiente: utilizamos una variable estática privada llamada "instance", que por defecto valdrá null. El constructor de dicha clase será privado, por lo que no seremos capaces de crear un nuevo objeto desde fuera de dicha clase.

Utilizaremos un método estático, llamado "getInstance" que será público. En este, preguntamos si la variable estática "instance" es null. En dicho caso, como estamos dentro de la clase seremos capaces de llamar al constructor y lograremos instanciar, además esa instancia será guardada en la variable estática. En caso contrario, si la instancia ya fue creada, la variable estática tendrá un valor distinto de null y se retornará siempre el mismo.

En nuestro proyecto, utilizamos este patrón en las clases Clinica y SalaDeEspera.

Patrón Factory

El patrón Factory facilita y centraliza la "responsabilidad" de crear instancias de objetos que tengan relación de herencia, es decir, que todas extiendan de una clase padre común. Mediante un método estático, crearemos las distintas instancias dependiendo lo se requiera. Para ello, como todas extienden de la misma clase, tendrán parámetros a la hora de su construcción comunes entre ellas. Además agregaremos dos parámetros, uno que identifique la clase que se quiere instanciar, normalmente utilizamos un String, y el otro parámetro será de tipo Object y se lo suele llamar "extra" que, básicamente, es una forma de generalizar aquellos atributos que sean necesarios por algunas clases además de los comunes a todas. En nuestro caso no fue necesaria la implementación del parámetro "extra", ya que todos los atributos eran iguales para todas las clases a crear.

Utilizamos el patrón Factory en dos ocasiones. Para la creación de los médicos, en conjunto con el Patrón Decorator que se explica más adelante. Principalmente usamos una clase *Medico*, de la cual se extienden tres clases: *MedicoCirujano*, *MedicoPediatra*, *MedicoClinico*. Entonces a la hora de crear un medico, delegamos esa responsabilidad al metodo *MedicoFactory*.

La segunda ocasión en la que le dimos uso al patrón fue para la creación de los pacientes, ya que tanto las clases Nino (evitamos la ñ), Joven y Adulto extienden de Paciente

Patrón Double Dispatch

El concepto de Double Dispatch se basa en la interacción entre un objeto y otro, cuando un resultado depende de ambos parámetros, y nos permite ahorrarnos varias líneas de código al evitar el uso de condicionales "if" anidados. Entonces, en lugar de utilizar un método sólo que resuelva dicha situación, conviene agregar nuevos métodos que tengan el mismo nombre para cada clase de todos los objetos que puedan ser parámetros.

Se escribe entonces un método que tenga como finalidad invocar a un método secundario del objeto parámetro. Ese método secundario es una operación específica sobre el receptor original. Por lo tanto va a ser responsabilidad de cada objeto que reciba esta invocación conocer qué debe hacer.

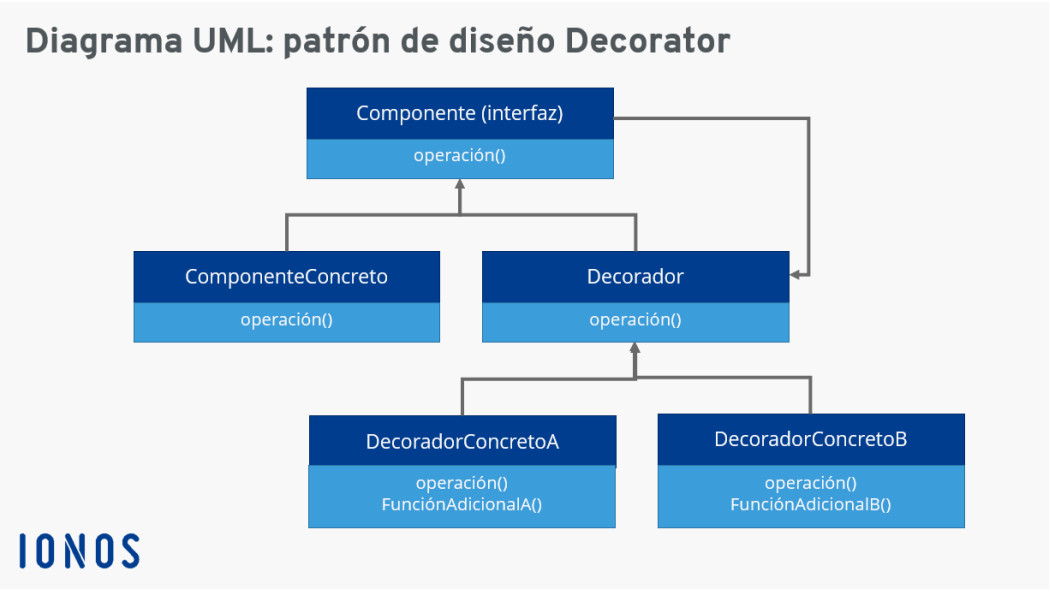
En nuestro ejemplo, decidimos utilizar el Double Dispatch para definir quién entra a la sala de espera privada, creando las clases Nino, Joven y Mayor. Estas tres, además de ser hijas de la clase Paciente, implementan Priorizable, interfaz que contiene los métodos que reescribirán cada una de las clases anteriormente mencionadas. El método principal es "prioriza" que recibe un priorizable como parámetro, y luego le delega a este el método secundario asociado al objeto sobre el que queremos trabajar.

Por ejemplo, tenemos un Niño en la sala de espera privada, desea ingresar un Joven, por lo tanto se invoca al método Nino.prioriza(Joven) que retornará el valor de la función Joven.ganaNino(). Esta última en este caso devuelve true porque el Niño le gana en prioridad de la sala al Joven, es decir, el Niño prioriza sobre el Joven.

Patrón Decorator

El patrón Decorator nos permite añadir funcionalidades y responsabilidades a un objeto dinámicamente durante la ejecución. Esto permite disminuir la cantidad de clases creadas para cada combinación posible, simplificando su codificación y siendo extensible. Para aplicar este patrón se crean clases decoradoras que implementan las funcionalidades que van a "decorar" al objeto.

Para poder aplicarlo correctamente cada clase decoradora tiene que ser del mismo tipo que el objeto concreto, por lo que se implementa una interfaz o clase abstracta que abarque tanto el objeto como la clase decoradora. La clase decoradora tiene un atributo del tipo interfaz (o clase abstracta) para que, por polimorfismo, delegue la ejecución del método al decorador concreto que se invoque en el momento.



Agregar una nueva funcionalidad sería tan sencillo como crear una clase decoradora concreta que sobreescriba el método a decorar y declare su comportamiento, sin necesidad de modificar ninguna otra clase.

Este patrón se puede complementar con el uso del Patrón Factory para poder delimitar y organizar la forma en la que se va decorando el objeto.

En nuestro proyecto este patrón fue utilizado para la creación de los médicos. Donde primero creamos un *MedicoBase* con su honorario básico y luego según su especialidad, contratación y posgrado vamos decorando el valor del honorario.

Cabe aclarar que la implementación de la clase abstracta *Medico* en un principio se planteó como una Interfaz, pero debido a que necesitabamos que los médicos decorados extiendan de la clase abstracta *Usuarios* optamos por usar una clase abstracta que extiende de la clase *Usuarios* para simplificar la cantidad de cambios a realizar.

Comentarios

Print de Factura

Al hacer que la Factura imprima su información, se quiso usar una librería externa (j-text-utils) que podía imprimir los datos como una tabla, funcionó pero algunos integrantes tenían problemas al agregar la librería al class path y entonces no podía correr el programa correctamente, se terminó optando por usar el método de System.out.Format para darle a la salida el aspecto de datos tabulados.