**AA3 PATRONES DE COMPORTAMIENTO:**

**1. PATRÓN COMMAND**

a) Descripción del escenario.

Una empresa de robótica nos pide que hagamos una aplicación para manejar robots (AppRobot). En un principio nuestra idea es implementar la clase robot y luego una clase que tuviera todos los métodos con las operaciones que puede hacer nuestro robot. Pero esto no es muy eficiente. Más tarde se nos ocurre que podríamos clasificar las operaciones secuenciales en varias clases para gestionarlas, pero esto tampoco es eficiente ya que si cambiamos algo de nuestra clase robot habría que cambiar todas las clases y sus métodos dependiendo de la lógica del negocio. Solución: encapsular las peticiones de las operaciones bajo algún método. Entonces creamos una clase que guarde las órdenes y otra clase que las ejecute, de forma que el cliente no sepa de las clases órdenes concretas las cuales estarían encapsuladas.

b) Diagrama de clases.

En el siguiente diagrama se puede observar un ejemplo de nuestra aplicación. Vemos cómo las clases OrdenMover, OrdenCoger y OrdenSoltar representan las órdenes concretas que implementan la interfaz OrdenRobot (que será el Comando) con el único método ejecutar (). La clase InvocadorOrden representará el Invocador el cual se dedicará a recoger las órdenes y almacenarlas en un ArrayList para luego pasárselas a la clase ReceptorOrden el cual las ejecutará realmente con el método ejecutar y el tipo de Robot que le pase el cliente.

Diagrama

Descripción generada automáticamente

c) Código fuente.

**2. PATRÓN MEDIATOR**

a) Descripción del escenario.

Una empresa de telecomunicaciones nos propone que hagamos una aplicación (iChat) para que varios usuarios se comuniquen entre sí. Al principio se piensa hacer la aplicación de forma que los usuarios se comuniquen directamente unos con otros. Pero con esto surge un problema: alto acoplamiento, el cual crea dependencias de unos con otros para que se puedan comunicar y que haya baja reutilización del sistema. Solución: aplicar el patrón Mediator, que hará que los usuarios no se tengan que comunicar directamente entre sí coordinando las comunicaciones entre ellos, encapsulando de esta forma el comportamiento de todos los usuarios en un solo objeto mediador, que controlará, coordinará y reducirá el número de comunicaciones entre usuarios y por lo tanto dejando un bajo acoplamiento.

b) Diagrama de clases.

A modo de ejemplo simple podemos ver en el siguiente diagrama, la interfaz Mediador y la clase abstracta Usuario (que representa al Colega) las cuales se hacen referencias mutuamente y las mismas son implementadas y extendidas por las clases UsuariosConcretos (1 y 2) y MediadorConcreto, respectivamente. Se puede apreciar que en ningún momento los usuarios se comunican entre ellos de forma directa, para conseguir esto, pasan la comunicación a través de la clase Mediador la cual pondrá en contacto a los usuarios, de forma indirecta, que quieran comunicarse.

Diagrama

Descripción generada automáticamente

**3. PATRÓN OBSERVER**

a) Descripción del escenario.

Una empresa de venta de aparatos electrónicos nos pide que realicemos una aplicación (NotifyProd) que avise a los clientes cuando el producto se encuentre en la tienda para que se acerquen a comprarlo. Nuestra solución es utilizar el patrón Observer, también llamado Publicador/Suscriptor y utilizaremos esta misma metodología para avisar a los clientes cuando se tenga el aparato requerido en la tienda. Tendremos tres tipos de mensajes para tres tipos de clientes según lo que se habló con ellos a la hora de meter sus datos en nuestra base de datos: unos que tardarán 1 o 2 días en ir a comprarlo, otros no más de una semana, y otros más de una semana.

b) Diagrama de clases.

En el siguiente diagrama podemos observar dos interfaces: una que representará al Sujeto llamada Publicador y otra llamada Observador que representará al Suscriptor. La interfaz Sujeto contendrá tres métodos sin implementar para que la clase SujetoConcreto las implemente, pudiendo añadir suscriptores, borrar suscriptores y mandar notificaciones a suscriptores que se encuentren en el ArrayList puesto aquí como atributo. La interfaz Suscriptor, que representará al Observador, tendrá un único método sin implementar: actualizar() el cual será implementado por las clases SuscriptorTipo1, SuscriptorTipo2 y SuscriptorTipo3 que implementarán este método cada uno a su forma con distintos mensajes ya que serán un distinto tipo de cliente cada uno.

