

**PROYECTO DE**

**DISEÑO**

**DE SOFTWARE**



Msc. David Jurado Mosquera

INTEGRANTES:

* Jácome Rodríguez David
* Paladines Bossano Jorge
* Serrano Herrera Ricardo

**DESCRIPCIÓN**

## Patrones de Diseño

Una vez analizando las especificaciones del sistema para la venta de productos de línea blanca, se determinó la posible implementación de varios patrones creacionales, estructurales y de comportamiento.

Específicamente, para el caso de los Productos como tal, el enunciado indica que actualmente trabajan únicamente con 3 productos distintos: lavadoras, refrigeradoras y cocinas, por tal motivo, se decidió implementar el diseño **Builder** para la creación de estos objetos, sin embargo, este patrón no es apto para el caso de que el local decida vender una gama más variada de productos, por lo tanto, el diseño del sistema sufrió un cambio en el cual se estableció el patrón creacional **Factory Method** como el ideal para el proceso de creación de los productos.

Los empleados tienen la opción de solicitar a los administradores tener más permisos en el sistema con la finalidad de poder modificar facturas y ventas. Para este proceso lo óptimo es utilizar el patrón de diseño estructural **Decorator** dado que es un proceso que se puede o no dar y simplemente añade una característica o funcionalidad al empleado.

Cuando el empleado solicita algún permiso adicional, este permiso debe ser notificado al administrador, si el administrador da el visto bueno, el empleado automáticamente obtiene todas estas ventajas, caso contrario, seguirá con sus funciones establecidas. Para ambas situaciones, se reciben notificaciones por parte de todos los involucrados cuyos datos son actualizados automáticamente, el patrón de diseño de comportamiento **Observer** facilita este comportamiento en el sistema.

Tanto los pagos como los reportes pueden ser generados de diversas maneras; en el caso de los reportes pueden existir: reportes de artículos, clientes y vendedores; en el caso de los pagos, estos se pueden realizar de dos formas: mediante visa o en efectivo. El patrón de diseño de comportamiento **Strategy** es el ideal para esta situación dado que las diversas formas de pago comparten un algoritmo similar para su realización, de igual manera sucede con los reportes.

Los clientes para ser atendidos deben esperar hasta que un empleado se encuentre libre y los pueda atender. Este tipo de comportamiento es característico del patrón de diseño de comportamiento **Chain of Responsibility**, por lo tanto, se implementó este patrón en el sistema.

Dada la cantidad grande de datos que se manejan, la compañía tiene su información en una base de datos, la cual es accedida durante gran parte del sistema para poder realizar consultas (entre otras funciones) del producto. Estas consultas no podrían ser realizadas si el sistema no cuenta con una conexión a la base de datos. Para esto necesitamos una única instancia que permita establecer dicha conexión. El patrón creacional **Singleton** es el ideal para esta situación.

Finalmente, dado que es un sistema manejado por varios usuarios cuyos roles establecen un nivel de acceso en el sistema, se debe manejar un log in, el cual, una vez recibidas las credenciales del usuario, establece qué parte del sistema se presentará. Para este caso se utiliza el patrón de diseño **Facade,** el cual se lo usa en conjunto con singleton para que se pueda llamar a la ventana desde cualquier parte y regresar a la selección de usuario rápidamente.

## Refactoring

* **Data Clumps**: Los métodos de ingreso para la base de datos, actualización y consultas se repetían en gran parte de las clases, se solucionó llevando cada método a una clase principal ya que al principio estos métodos estaban en cada una de las claves FXML.
* **Long Method:** Inicialmente nuestro método accionCrear estaba implementado en todas las clases UI y era un método de aproximadamente 25 líneas la cual la reducimos considerablemente y movimos este método a SuperAdmin ya que este es el que se encarga de la administración de los usuarios.
* **Long Class:** La clase Admin se encargaba de demasiadas cosas, así que la dividimos en partes que se encarguen de tareas más específicas. Esto también se dio en otras clases pertenecientes a Modelos en MVC.
* **Shotgun Surgery:** Se movieron las vistas al paquete view y por tanto se tuvieron que realizar varios cambios en cada uno del controlador de las vistas.
* **Lazy Class:** Se tuvieron que eliminar algunas clases de la implementación pues al momento de generar vistas y enlazar a la base de datos, algunas de las clases que quedaron ya no tenían funcionalidad, así que se las eliminó. Tal es el caso de “EmpleadoInterface.java” en el paquete Decorator.

# Tipos de comprobaciones

1. assertEquals() debe comprobar que cuando se ingresa el usuario, este sea el correcto.
2. Para la verificación de la contraseña se comprobaría con assertEquals().
3. Al momento de usar Factory Method para crear un producto, se comprueba con assertSame() que sea el mismo producto dado uno de prueba.
4. Para la conexión de la base de datos se utilizó el tipo de comprobación assertNotNull().
5. Para guardar los datos en la base de datos se utilizaría assertNotNull().
6. Cuando se regresa a la ventana de Login, se prueba con assertSame() que sean la misma ventana para comprobar que efectivamente se regresó al lugar correcto.
7. En la consulta de productos, assertNotSame() comprobaría que en ningún momento hayan 2 productos iguales.
8. Tanto en la creación de productos como en la de usuarios, fail() ayuda a comprobar que si se ingresa mal, arroja las excepciones que debe.
9. Durante la modificación de productos, antes de que se inserte la actualización, se puede usar assertSame() para asegurarse de que se esté modificando el mismo producto.
10. Luego de insertar varios productos, assertEquals() se encarga de que los productos colocados, puestos en un arreglo, sean iguales a unos predeterminados guardados en una lista de prueba, usados para la prueba unitaria.