Diseño del Sistema con Patrones

**Funcionalidad:** Asignar Privilegios

**Patrón:** Strategy

**Explicación:** Permite que el comportamiento de la asignación de privilegios sea determinado dinámicamente sobre un objeto base que es la clase PrivilegiosUsuarios. Por lo cual encapsulamos las clases para que sean intercambiables y reusables. Nos ayudará en el momento en que ya se asignó un privilegio pueda existir definir el comportamiento que tendrá el usuario en base a los privilegios.

**Funcionalidad:** Asignar Vendedor

**Patrón:** Chain of Responsibility

**Explicación:** Cuando un cliente entra al local se le debe inmediatamente asignar un vendedor que esté disponible para atenderlo. Este patrón nos ayuda ya que sirve para establecer una cadena de objetos que con una clase como emisora de la petición busque a través de una condición asigne un objeto de la cadena. Entonces este ayudará al momento que exista una alerta de cliente buscar de una manera más efectiva cual es el vendedor que lo atenderá.

**Funcionalidad:** Búsqueda de Artículos

**Patrones:** Strategy

**Explicación:** Este patrón servirá para esta funcionabilidad porque cada tipo de búsqueda contiene un query diferente para buscar en la base de datos, lo cual al buscar cada por cada sección se comporta de manera diferente cada búsqueda por lo tanto esa sección del programa.

**Funcionalidad:** Reportes

**Patrones:** Strategy

**Explicación:** Para la creación de reportes se usará este patrón ya que cada vez que se va a ver o crear un reporte se debe designar que tipo de reporte el usuario necesita y por cada tipo de reporte el programa tiene un query en la base diferente para que extraiga la información que se necesita mostrar en la tabla Reporte.

**Funcionalidad:** Pedir autorización

**Patrones:** Observer

**Explicación:** Este patrón sirve cuando se necesita que un objeto necesite notificar a otro en un cambio de estado. En el caso de nuestro programa lo utilizaremos ya que el vendedor puede pedir nuevos privilegios al administrador y a ese le llega una notificación para aceptar o rechazar su petición. En este caso el administrador será el observador y el vendedor es el que debe notificar su requerimiento de cambio de estado por la asignación de privilegios.

**Funcionalidad:** Pagos

**Patrones:** Strategy

**Explicación:** Cuando se necesita realizar un pago dependiendo de qué tipo va a utilizar el programa va a tener que realizar el pago de una manera distinta, ya que los tipos de pago tienen formas diferentes de realizarse.

**Funcionalidad:** Inicio

**Patrones:** Facade

**Explicación:** Este patrón es útil cuando se quiere crear una interfaz amigable con el usuario para poder crear diferentes subsistemas dependiendo de la necesidad del cliente. Los subprogramas de nuestra implementación son los diferentes vistas de usuarios que existen, ya que dependiendo de que usuario entra al sistema general se activará solo ese subsistema que es donde el cliente va a navegar.

Refactoring

**Clase:** PrivilegiosController

**Code Smell:** Large Class

**Refactoring:** Extract Class

**Explicación:** La clase PrivilegiosController, contenía todos los métodos que se utilizaban para asignar los privilegios en cada menú de los usuarios dependiendo de cual es. Se creó con Extract Class, una clase por cada tipo de privilegios que los calificamos dependiendo del menú (Productos, Ventas, Clientes, Cotizaciones, Usuarios). Estas clases están ligadas como herencia a PrivilegiosController para poder llamarlas dependiendo de los privilegios que tiene cada usuario, usa sus métodos.

**Clase:** Subclases de BusquedaArticulo

**Code Smell:** Long Method, Duplicate Code

**Refactoring:** Extract Method, Pull up Field

**Explicación:** Se identificó que las Subclases de BusquedaArticulo contenían un método buscar el cual era muy largo y realizaba funciones que no le correspondía que era conectar la base con el programa para obtener la información para realizar las diferentes búsquedas. Por lo cual se creó una clase llamada Conexión Model, que le agregamos el método conectar que solo realiza la conexión y en BusquedaArticulo que es abstracta se agregó el método ObtenerSet para obtener la información. Y en cada tipo de BusquedaArticulo (Categoría, Nombre y Descripción), se le agregó el método buscar y llenarSet para poder ingresar la información.

**Clase:** Reporte

**Code Smell:** Large Class

**Refactoring:** Extract Class

**Explicación: :** Se revisó las clases en el paquete View que correspondían a la sección de reportes, donde la clase era muy larga, y era posible la creación de nuevas clases a partir de los métodos que existían en la clase, ya que al seleccionar una opción en el combobox se desplazaban diferentes funciones que se presentarían en un TableView, y así dependiendo de las opciones se presentaría el reporte correspondiente.

**Clase:** Menú de Usuarios

**Code Smell:** Large Method

**Refactoring:** Extract Method

**Explicación:** Se revisó las clases en el paquete View que correspondían a los menú de usuarios, cada uno tenían un método organize() que colocaba los botones para el menú de los usuarios. Ese método crecía dependiendo de cada botón que se creaba, así que se decidió extraer el método, en métodos que retornaban un VBox que obtenían los botones para ingresar a las funciones, y que en organize solo llame a los métodos y agregue al Pane principal todas las funcionalidades. No se realizó una Extract Class porque nos había forma de extraer en otras clases. **:**

**Clase:** ReporteController

**Code Smell:** Duplicate Code

**Refactoring:** Extract Class

**Explicación:** En esta clase existía el mismo problema de búsqueda artículo, tenía un método que realizaba la conexión, la extracción de información de la base y el llenado de información. Entonces usamos las funciones que ya están creadas en la clase Conexión, para que haga la conexión a la base y realice la recolección de información necesaria dependiendo de qué tipo de reporte se quiere crear.