



**ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL**

**FACULTAD DE INGENIERÍA EN ELECTRICIDAD Y COMPUTACIÓN**

DISEÑO DE SOFTWARE

PROYECTO FINAL

DISEÑO DE UN SISTEMA PARA UNA APLICACIÓN

**INTEGRANTES:**

DEL PINO PAREDES GABRIEL ANDRÉS

FLORES SOLÓRZANO JUAN JOSÉ

LASO DELGADO JUAN FRANCISCO

HERRERA CRUZ ANTHONY KEVIN

**PROFESOR:** Msc. DAVID JURADO

**PARALELO:** 1

**TÉRMINO:** 1S-2018

Introducción

El diseño de software se basa en la identificación, evaluación, validación y especificación de los diferentes modelos requeridos para la implementación de software teniendo en cuenta los requerimientos funcionales y no funcionales. Un diseño es la base y el principio para realizar cualquier programa o aplicación. Dependerá del diseño tanto la implementación como el mantenimiento del software, es por eso, que se considera que un buen diseño no solo asegura una buena implementación sino asegura que sea necesario muy poco mantenimiento para el futuro. Para este proyecto se nos ha pedido el diseño de una aplicación para el manejo de inventario, compras y ventas de artículos. Utilizando la arquitectura MVC, y los paradigmas de programación que considere necesarios. El sistema debe cumplir con diferentes características.

Antecedentes

La empresa Línea Blanca S.A. ha decidido empezar sus actividades comerciales en el país y han elegido la ciudad de Guayaquil para poner sus locales comerciales. Esta empresa se dedica a la compra y venta artículos y electrodomésticos, ellos desean trabajar con proveedores nacionales obteniendo un descuento como mayorista y a su vez vendiendo en lugares estratégicos para aumentar la rotación de los artículos. A su equipo de desarrollo se le ha pedido que realice un diseño del sistema, incluyendo la base de datos y los diagramas de clases, de casos de uso, entre otros. Sin embargo, hay un par de detalles importantes acerca de cómo debe funcionar el sistema: El presidente de la compañía ha decidido que cada local / bodega no debe tener acceso a Internet. Además, el sistema debe permitir importar y exportar fácilmente la información en modo consola y con interfaz gráfica. El resto de las funcionalidades debe realizarse con interfaz gráfica.

Descripción

Nuestra propuesta de diseño se basa en distintos diagramas que permiten lograr entender lo que se desea implementar. En los diagramas UML se encuentran los diagramas de clases y aspectos, diagramas de casos de uso, diagramas de secuencia y, además el diagrama entidad-relación que es necesario para una futura implementación de la base de datos. Se ha escogido el paradigma orientado a objetos, ya que pensamos que es la mejor manera de plantear una implementación para nuestro programa. Siendo más específicos, nuestra solución se basa en tres pilares fundamentales: Un artículo, un empleado y un cliente. A partir de ellos y de elementos como el local, el inventario y las distintas transacciones, es que se van a diseñar las posibles relaciones que se pueden dar en el negocio.

DESARROLLO

Patrones de diseño escogidos

* Creacionales:
* FactoryMethod:

Se decidió usar este patrón para manejar una mejor manera de crear los distintos artículos que ofrece el sistema. En un principio se había optado por la opción de usar el patrón Builder, sin embargo, después de un análisis de todo el grupo y además la recomendación del profesor, se ha decidido finalmente que FactoryMethod es la mejor opción para este diseño de sistema.

Lo que nos pide la empresa consiste básicamente en crear artículos de línea blanca ya predefinidos, tal es el caso de lavadoras, refrigeradoras y cocinas de inducción. El patrón escogido permite realizar este requerimiento mediante el uso de una interfaz que permite crear objetos de una superclase, pero además permite a las subclases alterar el tipo de objeto que será creado. Esto no solo tiene un efecto positivo en el presente, sino además en un futuro, ya que suponiendo que la empresa decida agregar otro tipo de artículo, por ejemplo, una licuadora, será mucho más sencillo y menos costoso agregar ese tipo de producto en nuestro diseño.

* De comportamiento:
  + Strategy:

Pagos: Se decidió usar el patrón Strategy para implementar el comportamiento de los pagos, ya sea por tarjeta de crédito (VISA) o pago en efectivo. El concepto de este patrón se basa en tener varios comportamientos y hacerlos intercambiables, es decir, que, en este caso, un cliente pueda pagar en algún momento con tarjeta de crédito y otro pueda pagar en efectivo. En conclusión, el cliente tendrá varias maneras de pago a su disposición.

Reportes: El mismo principio que los pagos, los reportes se generan independientemente de los usuarios por lo que Strategy cumple el requerimiento.

* + Chain of responsibility:

Se utiliza este patrón para diseñar el comportamiento de la relación cliente-vendedor del sistema. Se solicita que un vendedor solo pueda atender un cliente, entonces, se debe buscar un vendedor disponible entre todos los existentes para poder asignarlo a un cliente y así empezar la atención.

La mejor opción de diseño para este caso es este patrón ya que este permite dar a más de un objeto la oportunidad de manejar una solicitud. La responsabilidad de esa solicitud va a ir cambiando de objeto hasta que uno de ellos maneje la misma. En nuestro caso, el vendedor si no está disponible pasará la responsabilidad al siguiente, hasta encontrar uno que sí lo esté, este se encargará de atender al cliente en espera.

Pruebas unitarias realizadas

1. AssertNotEquals : ControllerAdmin

Prueba para verificar que el resultado de la consulta de artículos de la base de datos no sea igual a null.

1. AssertNotNull : ControllerAdmin

Prueba para verificar que el resultado del conteo de artículos de la base de datos no sea null.

1. AssertEquals : ControllerGerente

Prueba para verificar que las referencias de los resultados de la consulta de ventas por mes sean las mismas.

Prueba para verificar que las referencias de los resultados de la consulta de clientes por mes sean las mismas.

1. AssertNotEquals : ControllerGerente

Prueba para descartar que las referencias de los resultados de la consulta de clientes no sean las mismas.

Prueba para descartar que las referencias de los resultados de la consulta de ventas no sean las mismas.

Prueba para descartar que las referencias de los resultados de la consulta de artículos no sean las mismas.

1. AssertNotNull : ControllerPrincipal

Prueba para verificar que el resultado del login usuario/contraseña no sea null.

1. AssertNotSame : CocinaInduccionFactory

Prueba para verificar que el resultado de una creación de cocina no tenga la misma referencia al resultado de otra creación de cocina.

1. AssertNotSame : LavadoraFactory

Prueba para verificar que el resultado de una creación de lavadora no tenga la misma referencia al resultado de otra creación de lavadora.

1. AssertNotSame : RefrigeradoraFactory

Prueba para verificar que el resultado de una creación de refrigeradora no tenga la misma referencia al resultado de otra creación de refrigeradora.

1. AssertTrue : VendedorFinder

Prueba para verificar que el resultado de una búsqueda de un vendedor libre sea verdadero.

1. AssertTrue : TarjetaVISA

Prueba para verificar que el resultado de la validación de credenciales de una tarjeta sea verdadero.

REFACTORING

* Code Smell: Comments >>> Refactoring: Extract Variable

Se ha encontrado bloques de código con comentarios explicativos, los cuales han sido eliminados y se han extraído ciertas variables para hacer entendible el código sin necesidad de comentarios.

* Code Smell: Primitive Obsession >>> Refactoring: Replace Data Value With Object

Se ha encontrado en clase Usuario y Cliente, que la cédula puede ser implementada como un objeto tipo Cédula y no como un String.

* Code Smell: Long Method >>> Refactoring: Extract Method

Se ha encontrado métodos muy largos en todo el paquete Vista, es por eso que se han extraído tales métodos en nuevos métodos que luego solo se llaman por el método original o por el constructor de la clase que implementa una Vista.