Proyecto 2 “Portal Vaquita”

Modelado y Programación

Vaquitas computitas

- 321088642 Flores Morán Julieta Melina
- 321244763 López Pérez Mariana
- 321102502 Zarco Romero José Antonio

Problemática

Numerosos estudiantes de la Facultad de Ciencias de la Universidad Nacional Autónoma de México (UNAM) se dedican a la venta de diversos productos dentro del campus universitario, que van desde artículos de papelería hasta alimentos en general; dicho emprendimiento les permite solventar sus gastos y continuar con sus estudios. Con el fin de promocionar sus actividades comerciales, el alumnado ha utilizado las redes sociales colectivas de la facultad. No obstante, la gran apertura de estas plataformas a todas las publicaciones ha generado una saturación de contenido sobre diversos temas, lo cual nubla la atención a los negocios.

Por otro lado, vale la pena mencionar el aumento significativo que tuvieron las aplicaciones de marketplace como Amazon, eBay, Walmart, Mercado Libre, entre otras, durante la pandemia del COVID-19. Estas plataformas jugaron un papel fundamental al facilitar la compra y venta de una amplia gama de productos de manera remota. Sin embargo, estas plataformas se enfrentan con dificultades para satisfacer las necesidades específicas de los estudiantes de la Facultad de Ciencias de la UNAM, siendo que como estudiantes nos es fácil interactuar y consumir productos disponibles en la facultad.

En este contexto, resulta pertinente que los propios estudiantes asuman el diseño de un marketplace adaptado a sus necesidades que identifican en su entorno. Por esta razón, se ha propuesto el desarrollo de un sistema destinado a facilitar el proceso de compra y venta entre

estudiantes. Este sistema sería de gran utilidad para los vendedores, ya que les permitiría no solo promocionar sus productos, sino también mantenerse actualizados sobre las transacciones y actividades relacionadas con sus ventas.

Implementación de cada patrón

Comportamiento

- Strategy

La decisión de emplear el patrón Strategy recae en implementar distintos medios de pago en la aplicación. Una vez terminada la compra, el cliente elige un método de pago entre los permitidos por el programa: Tarjeta, Efectivo y Transferencia. Cada uno de estos métodos implementa su manera de cobro definida por la interfaz FormaPago.java.

El uso del patrón beneficia la flexibilidad y extensibilidad del código, ya que coloca cada implementación en una clase separada donde el usuario puede intercambiar entre cada uno de los algoritmos de las formas de pago. Además, permite la personalización de campos de registro para los datos de pago.

- Iterator

Como es costumbre, decidimos utilizar el patrón Iterator para simplificar la manera en que recorremos los elementos de las estructuras de datos que almacenan los productos y críticas de los vendedores, así como los elementos y carritos de compras de los clientes.

Dentro de nuestro modelo definimos varios métodos que regresan un Iterable para trabajar con un punto de acceso secuencial de solo lectura que oculta la representación interna del inventario, el carrito, etc.

- Observer

Decidimos emplear el patrón Observer para varias clases del proyecto, entre ellas ServidorBaseDeDatos.java que implementa la interfaz Sujeto.java, cada vez que el servidor cambia, lo notifica a sus EscuchaServidor.java. El escucha concreto Bitacora.java reacciona ante

esta notificación ejecutando su principal comportamiento, que es llevar un registro en la salida estándar; esta clase bitácora no está acoplada a la clase servidor y puede reutilizarse en otros contextos de ser necesario.

Otro ejemplo donde utilizamos Observer fue para manejar los objetos de tipo Conexion.java en el ServidorBaseDeDatos.java, mismo que maneja una lista de conexiones que los conecta y desconecta. Sin embargo, el método notificar no está definido explícitamente, puesto que la manera en la que se notifican los cambios varían en base a la acción realizada, la cual es ejecutada en base al Mensaje.

Además, para esta aplicación es crucial que la información de las bases de datos, como los productos y críticas de los vendedores esté siempre actualizada. El patrón Observer permite que los clientes reciban notificaciones instantáneas sobre cualquier cambio en la base de datos del servidor, lo que garantiza que siempre tengan acceso a la información más reciente. Así, se actualiza la información en el disco duro, posteriormente en la copia local de cada conexión y por último se refleja el cambio en la interfaz gráfica.

Estructurales

- Proxy

El patrón de diseño Proxy ofrece una solución robusta y flexible para la seguridad de los administradores, vendedores y clientes de la aplicación. Por esta razón, decidimos emplear el mismo, y así, verificar y autorizar las solicitudes de los clientes antes de pasarlas al servidor real; por ejemplo, al iniciar sesión, al terminar una compra, etc. Esto permite proteger los datos y prevenir accesos no autorizados.

Por ejemplo, MercaditaProxy.java controla el acceso del objeto original, Mercadita, permitiéndole crear un objeto específico (Vendedor, Cliente, Administrador), al momento de obtener el objeto Usuario de su base de datos de usuarios.

Creacionales

- Singleton

Con el objetivo de garantizar una única instancia para la clase ControladorClimas.java del paquete mx.unam.ciencias.modelado.proyecto2.api, utilizamos el patrón Singleton con un método sincronizado de bloqueo comprobado doble. Utilizamos Singleton pues requerimos que la variable del controlador se comparta entre todos los hilos para manejar fácilmente el comportamiento de la API, es decir, queríamos mejorar el rendimiento del programa al evitar un consumo de recursos innecesarios por el coste de creación del objeto.

Modelo-Vista-Controlador

El presente sistema implementa el patrón Modelo-Vista-Controlador (MVC) en una aplicación JavaFX con el fin de facilitar la organización del código y mantenibilidad del mismo. La división de responsabilidades sigue la estructura:

- **Modelo**: Contiene la lógica de la aplicación y los datos. En este caso, serían las clases en el paquete mx.unam.ciencias.modelado.proyecto2. Estas clases representan los objetos como Cliente, Vendedor, Producto, etc.
- **Vista**: Contiene la interfaz de usuario de la aplicación. Estos son los archivos FXML en el directorio src/main/resources/fxml. Cada archivo FXML define la estructura de una escena en la aplicación.
- **Controlador**: Contiene la lógica que conecta el modelo con la vista y maneja las interacciones del usuario. Las clases controladoras se encuentran en el paquete mx.unam.ciencias.modelado.proyecto2.igu. Cada controlador se asocia con un archivo FXML y manipula los elementos de la interfaz de usuario definidos en ese archivo.

Extras

Sockets

Consideramos fundamental el uso de sockets para implementar la comunicación cliente-servidor del marketplace. En este caso, cuando un cliente modifica su copia en memoria de la base de datos, puede enviar rápidamente esa información al servidor para que se

actualice y se guarde en el disco duro. En consecuencia, el servidor puede distribuir esa información a los demás clientes de manera eficiente a través de los sockets.

Threads

A fin de lidiar con múltiples clientes y el “problema de lectores y escritores” decidimos emplear hijos de ejecución. Así, en el hilo principal de la aplicación agregaremos las conexiones de los usuarios y, cada que un usuario se conecte lo envolveremos en un hilo particular. De esta manera, el servidor se encargará de aceptar conexiones mientras continúe en ejecución y, dichas conexiones también dispararán un hilo para procesar en un ciclo los mensajes que reciban del flujo de entrada del enchufe.

Esto nos permitirá implementar un protocolo de sincronización (con ayuda de la keyword “synchronized” de java) que asegure las acciones de cada usuario, de tal manera que al modificar registros en la base de datos haya total exclusión mutua, pero al leer solo se excluyan aquellos hilos que quieran escribir, pero no aquellos que lean.

GUI

Se optó por utilizar la **biblioteca JavaFX** como tecnología para desarrollar la interfaz gráfica de usuario (GUI) de este sistema de marketplace estudiantil. La elección se debe a su riqueza en componentes y controles, que nos permiten personalizar los diálogos, ventanas y demás componentes de las vistas de la aplicación.

Las bibliotecas de apoyo de JavaFX se encuentran especificadas dentro de las dependencias del archivo pom.xml, así como la **biblioteca FontAwesome** que utilizamos para utilizar iconos en las vistas.

Para el desarrollo de estas vistas fue utilizado el **IDE Netbeans** como una integración con **Scene Builder** ya que esta herramienta nos permitió dibujar de forma más ágil el diseño de JavaFX dentro de la aplicación.

Lectura y escritura de archivos

La lectura y escritura de archivos es empleada para cargar y guardar la base de datos del sistema desde el disco duro del servidor. Los beneficios que trae simular un gestionador de

base de datos de esta manera, incluye la persistencia de datos, la portabilidad de la aplicación, la facilidad de implementación y el control directo sobre el formato de los datos.

Se utilizan cuatro archivos de texto que serian sus objetos en una línea de texto, para ello deben sobrecargar los métodos de la interfaz Registro.java.

- usuarios.txt: Guarda la información de los usuarios del sistema.
- productos.txt: Guarda la información de los productos de cada vendedor del sistema.
- criticas.txt: Guarda la información de las críticas de los vendedores del sistema.
- carritos.txt: Guarda la información de los carritos de compras de los clientes del sistema.

Pruebas unitarias

Con el fin de asegurar que cada componente del sistema cumpla con sus requisitos y funcione correctamente, nos encargamos de realizar pruebas unitarias de cada módulo con el mayor aislamiento posible; dichas pruebas se realizaron con la biblioteca de Java **JUnit** y se encuentran en los paquetes mx.unam.ciencias.modelado.proyecto2.test y mx.unam.ciencias.modelado.proyecto2.red.test.

Al tener pruebas unitarias que cubren cada módulo, se facilita la refactorización del código, esto permite detectar y resolver errores de manera más eficiente, sin afectar el funcionamiento del sistema en su conjunto.

Interfaz de Programación de Aplicaciones (API)

Decidimos integrar una API en el marketplace, con el fin de proporcionar un valor agregado como el pronóstico del tiempo, con ella se pueden personalizar notificaciones para los usuarios, por ejemplo, brindar la capacidad de enviar alertas específicas sobre lluvias o frío a los estudiantes.

Con el objetivo de proporcionar datos meteorológicos confiables y actualizados, el sistema se apoya de la interfaz de programación de aplicaciones de la página del Servicio Meteorológico Nacional (SMN) de la Comisión Nacional del Agua (**CONAGUA**) de México.

- https://smn.conagua.gob.mx/es/web-service-api

La implementación de la API se encuentra en las clases del paquete mx.unam.ciencias.modelado.proyecto2.api. Además, para completar la conexión con la API nos

apoyamos de la **biblioteca Jackson** para manejo de JSON, especificada en el archivo pom.xml.

Comentarios

1. Sistema de construcción de software. El sistema de construcción de software que utilizamos para el presente proyecto fue **Maven**. Nos apoyamos de Maven, ya que facilita en gran tamaño cargar bibliotecas que no forman parte de la edición estándar de Java, entre ellas JUnit y JavaFX a través del modelo de objetos del proyecto, pom.xml. También, nos ayuda a generar la información del proyecto como la documentación **Javadoc** y el informe de pruebas unitarias.

2. Puerto & IP. Es importante mencionar que para fines prácticos y, dado que el programa se ejecuta en una sola máquina, utilizamos el puerto **1234** y la IP **localhost** al momento de instanciar:

    a. El Socket en el método conectar() de Mercadita.java.
    b. El ServerSocket en el constructor de la clase ServidorBaseDeDatos.java en el paquete mx.unam.ciencias.modelado.proyecto2.red.

Por lo que si dicho puerto se encuentra ocupado se producirá un error al intentar iniciar el programa. Es fundamental asegurarse de que el puerto especificado esté disponible y no esté siendo utilizado por otro programa. En caso de que el puerto 1234 esté ocupado, se puede optar por cambiar el número de puerto en las clases mencionadas anteriormente para evitar conflictos y permitir que el programa se inicie correctamente.

3. Dinero del Usuario. A diferencia de la forma en la que guardamos los registros de usuarios, productos, criticas y carritos, consideramos que manejar las cuentas bancarias corresponden a un servicio externo, como lo es un “banco”, lo cual está fuera del alcance de nuestra problemática.

4. Carritos de compras. Otra consideración que tener es con la forma en que se guardan los carritos de los diferentes usuarios. Cuando un usuario va llenando su carrito de compras, este se guarda y está disponible durante su sesión, y se guarda permanentemente en memoria al momento de cerrar sesión, para facilitar y mejorar la experiencia de compra. Pero por esta condición, si se cierra el

programa abruptamente las actualizaciones al carrito de compras no se guardaran, se tiene que cerrar sesión apropiadamente.

5. Error de conexión. En caso de presentarse un error entre la comunicación cliente-servidor, el servidor desconecta inmediatamente la conexión de su lista con la justificación de prevenir futuros errores. Dicha acción es notificada en la bitácora del servidor, sin embargo no se despliega en la interfaz gráfica del usuario.

6. UML. El editor de diagramas Dia no nos permitió ajustar todos los recuadros y hay algunos pequeños desbordes cuando contienen textos largos.

7. Directorios.

   Nuestro proyecto tiene dos carpetas principales:

   - src: Decidimos dividir el contenido principal de nuestro proyecto de la siguiente manera:
     - main: Contiene lo necesario para usar nuestro programa.
       - resources: Contiene los archivos Fxml que maquetan nuestras vistas y las imágenes que usamos en la interfaz.
       - java: Contiene el código fuente de nuestro proyecto y se divide finalmente en tres:
         - api: Separa las clases que se encargan exclusivamente de la comunicación con la api de la conagua y el manejo de la afirmación obtenida.
         - igu: Contiene todos los controladores y archivos relacionados con las vistas de JavaFx.
         - red: Contiene el código relacionado a manejar la dinámica de cliente-servidor
     - test: Contiene las pruebas unitarias que realizamos, separando las pruebas de red.
   - bin: Contiene los scripts para iniciar el servidor y el cliente.

Ejemplos de Entradas

El sistema únicamente cuenta con los siguientes usuarios registrados:

ID	ROL	NOMBRE_ USUARIO	CONTRASENA	TELEFONO	NUMERO_CUENTA
1	cliente	SusanaD	1234	5571380253	1234567890123456
2	cliente	Mario	1234	5520848819	6789012345678901
3	vendedor	Laura	1234	5529311982	2345678901234567
4	vendedor	Antonio	1234	5525337733	7890123456789012
17	administrador	Tamara	1234	5525337733	4567890123456780

No existe manera de registrar un usuario en el programa, solamente de iniciar sesión con los usuarios previamente cargados.

Para realizar pagos con tarjeta se necesita la información de una cuenta bancaria existente y asociada al usuario guardada en el banco, que son los siguientes:

NOMBRE_ USUARIO	NUMERO_CUENTA	NOMBRE	CVV	SALDO
SusanaD	1234567890123456	Susana Duarte	123	8000
Mario	6789012345678901	Mario Lopez	321	3000

Ejecución

Este proyecto fue desarrollado en **Fedora 38** y java version 17.0.9

Dentro del directorio proyecto2/ del presente zip, se encuentra el archivo pom.xml, al colocarse ahí y ejecutar el comando mvn install se compila el código fuente, se ejecutan las pruebas unitarias, se empaqueta el proyecto y luego se instala el artefacto resultante en el repositorio local de Maven.

Para correr el programa primero se tiene que ejecutar el **script bash** ./bin/servidor-proyecto2.sh, que pone en funcionamiento el servidor del programa, posteriormente en otra terminal, se debe ejecutar el script bash ./bin/cliente-proyecto2.sh, que lanza la aplicación con interfaz gráfica para los usuarios.