UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Modelado y programación.

Rosa Victoria Villa Padilla

[2025-1] (20/10/2024)

Proyecto 1:

CheemsMart.

Equipo: Christian.

- Leon Navarrete Adam Edmundo
- Rubio Resendiz Marco Antonio
- Valencia Pérez Guillermo Emanuel

Ciudad Universitaria, Cd. Mx. 2024

Anotaciones sobre la estructura de la práctica.

El proyecto consistió en emplear los conocimientos adquiridos durante el curso para diseñar un sistema de compra y venta de productos para una tienda de nombre CheemsMart.

Elección de patrones de diseño.

Strategy.

El uso del patrón Strategy en nuestro proyecto permite el manejo flexible y escalable de dos estrategias importantes para el funcionamiento del programa: Monedas e Idiomas. De esta forma cada una de las estrategias estará encapsulada en clases separadas, facilitando que estas sean asignadas en tiempo de ejecución según el país del cliente que esté usando el sistema sin afectar el flujo de este.

- Idioma: Cada idioma a implementar en el sistema se basa en una interfaz de nombre Idioma, que en general define una tira de métodos que devuelven cadenas para cada acción que pueda realizar el usuario al usar un menú, así como la forma visual bajo la cual se muestran sus opciones y el resultado final de su compra (ticket). Para esta interfaz se implementaron 3 idiomas: Español, Inglés y Portugués.
- Moneda: Cada divisa que pueda ser usada en el sistema se basa en la interfaz de nombre Moneda, que define un método para identificar la divisa (mediante su código ISO) y una forma de calcular el precio del producto a partir de una entrada numérica (que para esta implementación consistirá en una conversión de dólares a cualquier otra divisa). Para esta interfaz se implementaron 3 divisas: Peso, Dólar y Real.

Decorator.

El uso del patrón Decorator en el proyecto permite un manejo acumulable y extensible para poder agregarle funcionalidades a cada producto sin necesidad de alterar directamente las clases concretas de un producto. Se escogió este patrón específicamente para simular un escenario análogo a como se hace uso de los descuentos por parte de las empresas, pues manejar descuentos de forma acumulativa no supone una suma de sus descuentos, sino una composición de un descuento una vez aplicado otro (el margen de ganancia de una empresa se verá afectado si dos descuentos del 15% suponen uno del 30%).

Para el sistema se hace uso de una interfaz de nombre Producto y dos clases que la implementan (una de ellas una implementación concreta y otra una decoradora). Además, cada Producto tiene asociada una instancia de una enumeración de nombre Departamento, para categorizar las secciones asociadas a cada producto.

- ProductoConcreto: La clase concreta de los productos, en general define los datos asociados al producto (código, nombre, precioBase y departamento) así como formas de modificar y acceder a esta información.

- ProductoDecorator: Clase abstracta bajo la cual se decoran los productos, tiene una implementación de la interfaz y un País que consiste en una región bajo la cual el descuento es aplicable. En general se sobrecargan los métodos de la interfaz a los métodos de la implementación que está asociada a la clase, con la excepción de los métodos para calcular el precio del producto, para mostrar sus detalles y para envolver a otro producto. Las tres clases concretas asociadas corresponden a descuentos del 15%, 25% y 50%.
  Nota: el método envolver() es análogo a una idea implementada en la segunda práctica del curso, donde una instancia del decorador puede decorar a otros productos dadas ciertas condiciones, de esta forma el proceso de decorado es interno y no será necesario que especifiquemos el nombre de la clase para decorar un producto, haciendo este proceso más flexible.

Builder.

Se empleó el patrón de diseño Builder para la creación de los Carritos de compra en tienda, de esta forma podemos manejar de manera flexible la construcción de este tipo de objetos. Esto se hizo principalmente para la creación de Carritos de forma incremental paso por paso a medida que el usuario agregue productos.

Además, para el enfoque que tenemos en el sistema, consideramos importante que los carritos creados sean inmutables, pues el objetivo es poder garantizar que estos objetos no experimenten cambios después de su creación para poder ser procesados por el sistema una vez se decida proceder al pago de los productos. Esto nos permite separar la lógica de construcción del objeto construido para restringir la parte dinámica del programa únicamente a lo que está previo al proceso de pago.

Para lo anterior se hace uso de la interfaz CarritoBuilder que define formas de agregar, eliminar y aplicar descuentos a los productos, así como una forma de construir un carrito, la implementación concreta tiene por nombre CarritoBuilderConcreto. Los carritos de compra son instancias de una clase concreta Carrito que solo cuenta con una lista de productos, un cliente y formas de acceder a su información.

Factory.

Se empleó el patrón de diseño Factory para la creación de Clientes y Productos contemplando la posibilidad de que eventualmente la forma de asignar u obtener la información de estas instancias pueda cambiar en el futuro.

Es importante considerar que nuestro sistema utiliza archivos para capturar la información de las instancias fabricadas, de ahí que se utilicen listas y arreglos de tipo String para obtener la información. De ahí que esté el paquete *data/*.

Además, una parte fundamental de las fabricas es que su objetivo principal es poder proporcionar un objeto iterable de los objetos fabricados.

Nota: Durante la implementación surgieron problemas con los genéricos de los iteradores que nos llevaron a realizar una homogeneización forzada de los objetos Cliente y Producto, pero a medida que se desarrollaba esta solución dimos con un diseño muy intrincado y poco extensible de las fábricas. De esta situación se llegó a la conclusión de que resultaría más claro segregar las fábricas en dos (es decir, usar el patrón dos veces) para evitarnos la necesidad de forzar una homogeneidad en los objetos por cada incompatibilidad de tipo. Esto es importante porque nos lleva a un diseño más cohesivo y especifico sobre la manera en la que se crean tanto los objetos fabricados como los iterables generados.

Ambas fabricas (ProductoFactory y ClienteFactory) cuentan con una forma de generar un iterable a partir de una lista de tipo String, dejando un método abstracto a implementar por las clases fabricantes que implementan una forma de fabricar los objetos (fabricaProducto() y fabricaCliente()).

Iterator.

Se empleó el patrón de diseño Iterator para los objetos iterables que se generan en las fabricas del patrón Factory. Se hizo de esta forma contemplando la posibilidad de que eventualmente se desee cambiar la estructura de datos utilizada sin la necesidad de alterar la forma en la que se generan los iterables dentro de las fábricas.

- Iterable: En general define una forma de agregar, eliminar, tomar y verificar la contención de un objeto (Producto o Cliente), además de implementar una forma de obtener un iterador de esta estructura. Las dos clases que implementan Iterable son: ProductoIterable y ClienteIterable.
- Iterator: Clases que implementan una forma de obtener un iterador de una estructura de datos. Para ello tienen que obtener una estructura iterable mediante el constructor. Implementa los método hasNext() y next(). Las dos clases que implementan Iterator son ProductoIterator y ClienteIterator.

Observer.

Se empleó el patrón de diseño Observer para proporcionar una forma de que los usuarios reciban notificaciones sobre descuentos en el sitio. Para nuestra implementación, las clases que fungen como Sujeto y Observador son las siguientes:

- Sujeto: La clase ClienteRemoto es el sujeto observable de nuestro sistema para proporcionar información a los usuarios en base a información que se obtuvo de un servidor. Es particularmente útil porque de esta forma podemos hacer que los observadores reciban notificaciones siempre y cuando hayan iniciado sesión en el sitio.
- Observador: La clase MenuCatalogo es el observador de nuestro sistema pues es la clase que interactúa directamente con el usuario, de esta forma toda información obtenida del

ClienteServidor es empleada por esta clase para mostrársela al usuario. De este modo recibe notificaciones del ClienteServidor en tiempo de ejecución.

Proxy.

Se empleó el patrón de diseño Proxy para plantear un modelo Cliente-Servidor para los usuarios que deseen utilizar el programa. En general el objetivo es que con un servidor que contenga toda la información que el cliente podría solicitar para el uso del programa. La interfaz que se utiliza para esta parte recibe el nombre de Catalogo.

- Servidor: La clase servidora CatalogoServidor cuenta con un iterable de productos y uno de clientes, así como una lista de los usuarios activos que han accedido a la información del servidor. Esta clase gestiona lo referente a inicios y cierres de sesión, así como una simulación de las ofertas disponibles y una forma de transmitirlas a los que soliciten acceso a esta información. Además, esta clase es un Singleton.
- Proxy: La clase proxy CatalogoProxy solamente proporciona una forma de comunicarse con un servidor, en general sobrecarga todos los métodos a los de este servidor. Funge como intermediaria entre el cliente y el servidor.
- ClienteRemoto: Clase a la que tienen acceso todos los usuarios que deseen acceder al programa, en general es una clase que conecta de forma indirecta con el servidor y conecta de forma directa con los Menus que son mostrados a los clientes. Esta clase es el sujeto de nuestro patrón Observer.

Singleton.

Se empleó el patrón de diseño Singleton para lo relativo al CatalogoServidor del patrón Proxy. De esta forma toda la información a la que los usuarios acceden es exactamente la misma, lo que nos ahorra conflictos o errores concurrentes. Además, de esta forma podemos llevar un control de los clientes que solicitan acceso a la información del servidor.

Respecto al uso del proyecto.

Usuarios:

Para usar el programa es necesario que consideren la siguiente tabla de usuarios ya cargados. Primero es necesario que proporcionen el usuario y después la contraseña.

ID	Nombre	Pais	Usuario	Contraseña	Dinero
0001	Marco Rubio	MEXICO	mrubio	PASS123	10500.75
0002	Guillermo Garcia	ESTADOS_UNIDOS	ggarcia	PASS456	8000.5
0003	Adam Smith	BRASIL	asmith	PASS789	5000
0004	Christian Johnson	MEXICO	cjohnson	PASS321	100000

Nota: Cabe aclarar que esta información se carga en el programa mediante el archivo *data/Clientes.csv*, si desean agregar un nuevo usuario, pueden hacero siempre y cuando se proporcionen los datos directamente en el archivo.

Para ejecutar el programa:

El proyecto fue desarrollado en la versión de java 11.0.22 y se empleó Linux (Debian y Ubuntu) durante su desarrollo. Para ejecutarlo en Linux (para distribuciones basadas en Debian) es necesario usar los siguientes comandos en terminal:

Cabe mencionar que para usar el programa es necesario que se utilicen dos terminales, una para encender el servidor de la tienda y otra para hacer uso del programa como un cliente. Es necesario que el servidor esté encendido para que se pueda usar el programa.

Para compilar todas las clases y generar los archivos .class

```
$ javac -d bin $(find src -name "*.java")
```

Para encender el servidor (Terminal 1):

```
$ java -cp bin mx.unam.ciencias.modelado.proyecto1.proxy.CatalogoServidor
```

Para ejecutar el programa como usuario, se realiza una simulación. (Terminal 2):

```
$ java -cp bin mx.unam.ciencias.modelado.proyecto1.Main
```

Importante: Una vez ejecutado el Main, el programa lanza la siguiente excepción:

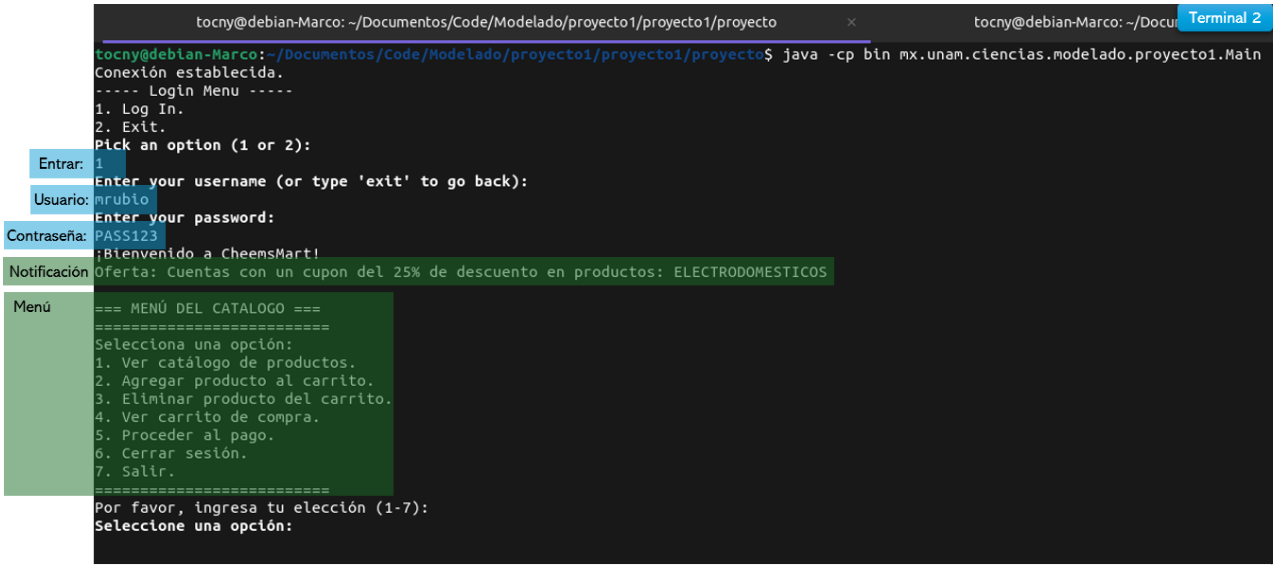
```
$ Error de comunicación remota: Comunicación interrumpida.
$ Error unmarshaling return header; nested exception is:
$    java.io.EOFException
```

Esto sucede debido a que, una vez que el servidor detecta que ya no hay más clientes accediendo a la tienda, el servidor cierra automáticamente. Esto se hizo con el único propósito de que el programa terminara.

Ejemplo de uso:

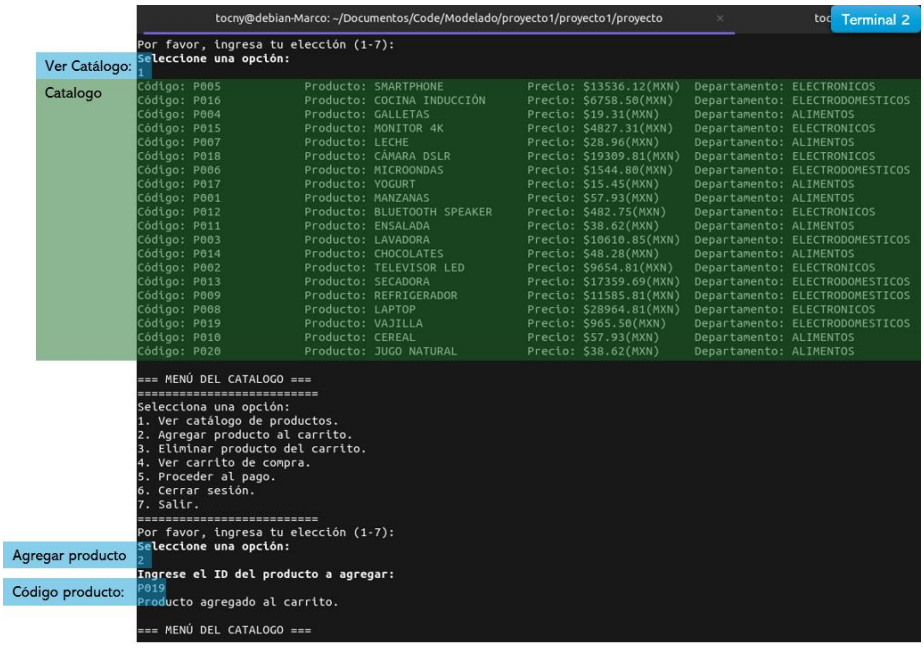
Compilar el programa y encender el servidor:

Iniciar sesión desde una cuenta:

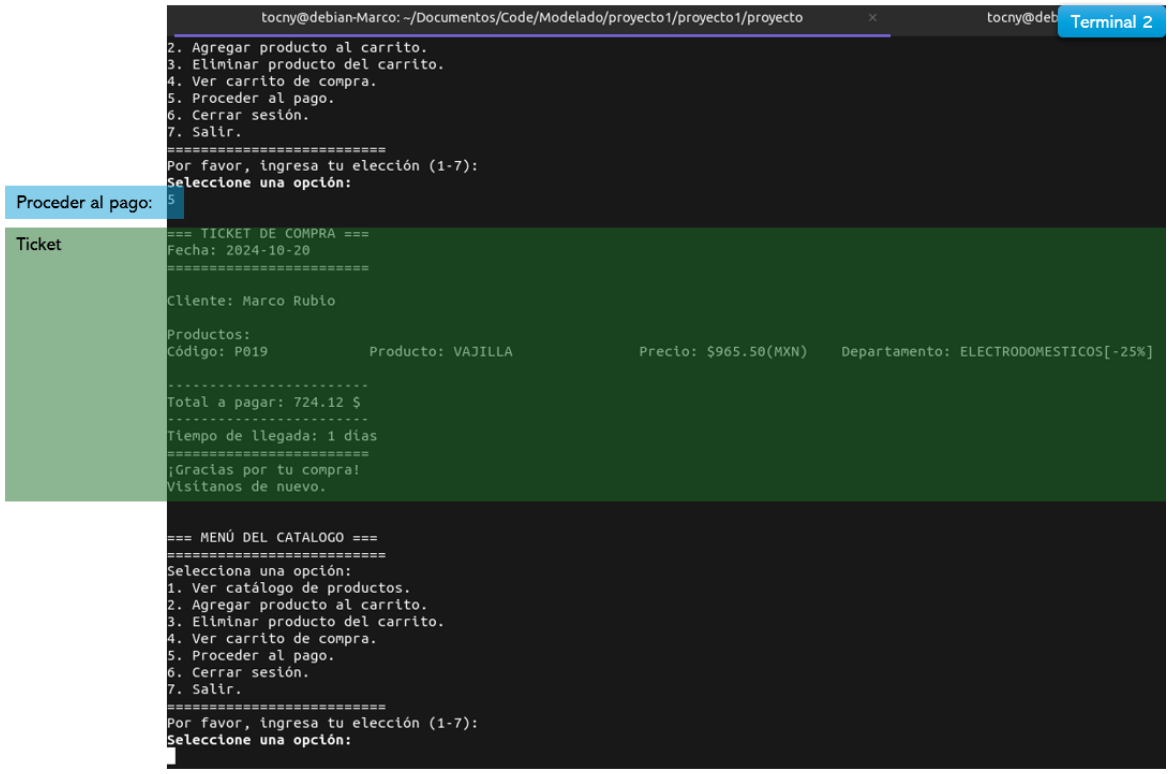


Inicio de sesión desde el servidor:

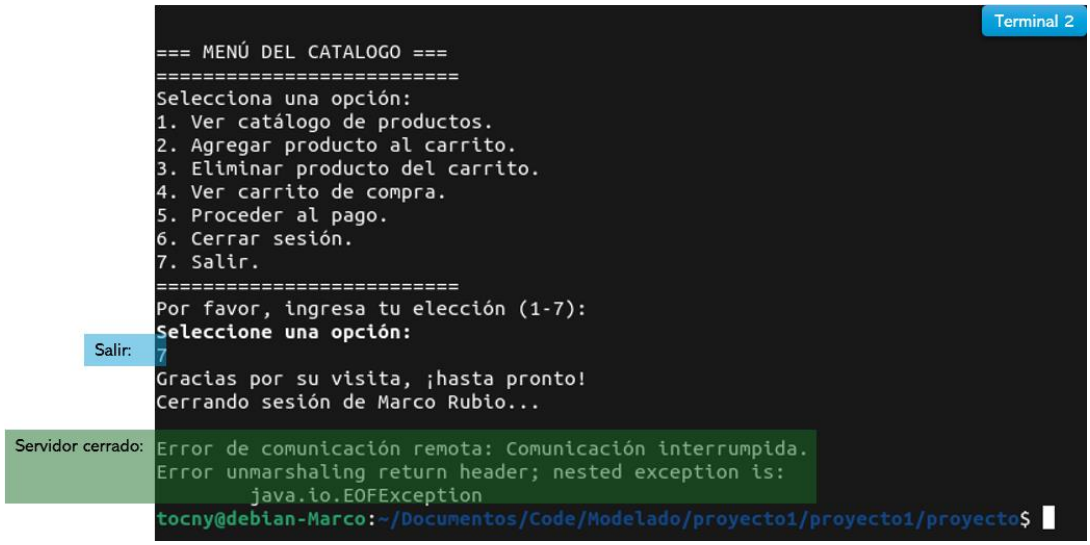
Interacción con el menú: ver catalogo y agregar producto.



Interacción con el menú: proceder al pago.



Interacción con el menú: Salir.



Cierre del servidor (por falta de clientes.):