Proyecto 01

Modelado y Programación

2025-2

Equipo LinuxCeros

- **Domínguez Sánchez Gustavo Alonso**

- Sistema Operativo: Fedora 38

- Versión de Java: OpenJDK 22.0.1

- Editor de codigo: Visual Studio Code

- Compilación y ejecución: Realizada desde terminal de Fedora utilizando `javac` y `java`

- **De La Cruz Reyes Jesús Ramón**

- Sistema Operativo: Fedora 41

- Versión de Java: OpenJDK 17

- Editor de codigo: Visual Studio Code

- Compilación y ejecución: Realizada desde terminal de Fedora utilizando `javac` y `java`

Informacion General

Este proyecto modela MonosChinos MX, una tienda especializada en el ensamblaje de computadoras . Debido al crecimiento acelerado de su popularidad, se ha planteado una expansión que involucra múltiples sucursales (Jalisco, Chihuahua, Yucatán) las cuales reciben pedidos y una sede central (CDMX) dedicada unicamente al ensamble y distribución de equipos.

Este proyecto tiene como objetivo el desarrollo de un sistema interactivo que permita a los clientes:

- Comprar equipos prearmados o personalizados
- Elegir entre una variedad de componentes individuales como CPU, RAM, etc.
- Instalar software adicional opcional como Windows, Office, AutoCAD o herramientas para desarrolladores
- Recibir un ticket completo con todos los detalles del pedido, incluyendo compatibilidad y precios

El desarrollo se realizó siguiendo los principios del patrón Modelo-Vista-Controlador (MVC), e integra múltiples patrones de diseño como Abstract Factory, Builder, Decorator, Adapter y Singleton para asegurar una arquitectura modular, mantenible y extensible.

Además de ofrecer un sistema interactivo con el usuario, internamente, el sistema permite simular pedidos de forma independiente mediante clientes ficticios, con lo cual se representa la concurrencia y se prueba la capacidad de la central CDMX para atender múltiples pedidos simultáneamente.

Descripcion del programa

Al iniciar el programa, el usuario es recibido con un menú principal que le permite seleccionar desde qué sucursal desea realizar su pedido. Las sucursales disponibles son: Jalisco, Chihuahua y Yucatán. Esta elección no altera la funcionalidad del sistema, pero sirve para simular el punto de origen del pedido dentro de la arquitectura de MonosChinos MX. Ademas se incluye la central CDMX, y como solo se dedica a ensamblar las PC, al elegir esta opcion permite ver todos los pedidos que ha procesado (antes y durante de la ejecucion actual del programa).

Una vez seleccionada la sucursal que pueda realizar pedidos (no central CDMX), se inicia el flujo de compra, el cual es dirigido por un controlador asociado a la sucursal. A través de una interfaz interactiva por consola, el sistema guía al usuario paso a paso para definir el tipo de compra y los componentes que formarán parte de su equipo.

El sistema permite dos modalidades de compra:

1. Equipo prearmado: El cliente elige entre configuraciones prediseñadas, pensadas para ciertos perfiles. Actualmente, se ofrecen los siguientes modelos:
    - Estudiante: componentes económicos y funcionales para tareas escolares.
    - Gamer: rendimiento medio-alto, balance entre costo y potencia.
    - Punto de Venta: configuración ligera, ideal para terminales de negocio.

Estas configuraciones ya están definidas en el sistema y se ensamblan automáticamente sin intervención adicional del cliente.

2. Equipo personalizado
   El cliente construye su PC desde cero, eligiendo uno por uno los componentes de cada categoría:
    - CPU, RAM, Placa Base, Disco Duro (HDD), Unidad de Estado Sólido (SSD), Fuente de alimentación, GPU, Gabinete.

Cada componente se selecciona desde un catálogo desplegado en consola. Para categorías como RAM, HDD y SSD, el cliente puede seleccionar más de una unidad, hasta un máximo de 4. Para los demás componentes, solo se permite seleccionar uno por categoría.

Una vez que el usuario termina de seleccionar sus componentes, el sistema verifica automáticamente si los elementos seleccionados son compatibles. En particular, si el cliente selecciona un procesador AMD, el sistema reconoce que no es directamente compatible con las placas base del catálogo (centrado en Intel), por lo que lanza una advertencia y la central CDMX realiza una adaptación automática mediante el patrón Adapter. Esta adaptación permite continuar el flujo sin errores ni interrupciones, dejando claro en el ticket final que se trató un caso de compatibilidad.

Después del ensamble de hardware, se ofrece al cliente la posibilidad de instalar software adicional. El cliente puede elegir entre:

- Windows 10 / 11
- Microsoft Office 365
- Adobe Photoshop
- AutoCAD
- Terminal de desarrollo (simulada con WSL)

Cada software puede seleccionarse solo una vez, y es completamente opcional.

Una vez ensamblada la PC, el sistema genera un ticket detallado con la siguiente información:

- Descripción completa de todos los componentes seleccionados
- Lista del software instalado
- Precio individual de cada componente y costo total
- Notas sobre adaptaciones de compatibilidad, si existieron

El ticket se muestra al usuario en consola y también se guarda en el archivo pedidos.txt para futuras consultas. Hasta este punto, se ha cumplido todo el flujo de interaccion con el usuario y por lo tanto la función principal del programa.

Adicionalmente, debido a la implementacion de Singleton, aprovechamos la oportunidad para el manejo de hilos: mientras el usuario interactúa con el sistema, en segundo plano se ejecuta una clase llamada `ClienteSimulado`. Esta clase representa pedidos ficticios realizados desde cualquier sucursal de forma aleatoria. Estos pedidos se construyen con datos generados automáticamente por una clase auxiliar llamada `DatosAleatorios` y se procesan de manera real, guardando sus tickets en el mismo archivo compartido Gracias al uso de hilos, el sistema puede manejar pedidos reales y simulados al mismo tiempo sin interferencias, demostrando un comportamiento concurrente simulando a un sistema real. Ya que esta funcionalidad extra al programa, desde el metodo `main`, se pueden inicializar mas clientes ficticios, o bien quitarlos por completo, sin afectar de ninguna manera la interaccion real con el usuario.

Patrones de Diseño

- AbstractFactory

El patrón Abstract Factory permite crear familias completas de objetos relacionados sin especificar sus clases concretas. En este proyecto, tenemos múltiples tipos de componentes (CPU, RAM, GPU, etc.), y cada uno tiene diferentes modelos, marcas y configuraciones. El uso de este patrón nos permite:

- Centralizar la creación de objetos de cada tipo de componente
- Desacoplar el sistema del conocimiento de las clases concretas
- Extender fácilmente la incorporación de nuevos productos (por ejemplo, nuevas marcas o modelos de componentes)

Creamos una jerarquía de clases donde cada tipo de componente tiene:

1. Una clase abstracta que define sus atributos y método `crearComponente()`.
2. Varias subclases concretas que implementan dicho método con valores específicos de cada modelo
3. Una fábrica concreta por tipo de componente (`FabricaCPU`, `FabricaRAM`, etc.) que implementa el método `getComponente(String tipo)`.
4. Una clase directora DirectorFabrica, que conoce todas las fábricas y permite construir cualquier componente del sistema desde una sola interfaz, ocultando los detalles al usuario.

De esta manera nuestras clases de AbstractFactory, convenientemente ubicadas en el paquete `modelo.fabricaabstracta` son:

- `CPU`, `RAM`, `GPU`, `SSD`, `HDD`, `PlacaBase`, `FuenteAlimentacion`, `GPU`, `Gabinete`: Clases abstractas de los productos base.
- `IntelCorei3_13100`, `Kingston8GB`, `NVIDIA_RTX4090`, etc.: Subclases concretas (productos reales)
- `FabricaCPU, FabricaRAM`, ... : Fábricas concretas
- `DirectorFabrica` : Clase directora que encapsula la lógica de construcción y delega a las fábricas correspondientes

Gracias a este patrón, pudimos construir un sistema flexible y extensible. evitando estructuras complejas de if o switch dispersas en todo el código para construir los componentes.

- Builder

El patrón Builder es ideal cuando necesitamos construir objetos complejos paso a paso, con configuraciones flexibles que pueden variar según las necesidades del cliente. En nuestro caso, una PC

personalizada o prearmada consta de múltiples componentes que deben ensamblarse de forma controlada y en orden.

Builder permite separar la lógica de construcción de la representación final del objeto (`PCBase`), lo cual:

- Facilita agregar o quitar componentes sin modificar la clase producto
- Permite tener distintos tipos de ensambles (personalizados o predefinidos)
- Encapsula el proceso de ensamblaje y permite reutilizar la lógica

En particular, aplicamos este patron creando una clase abstracta `BuilderPC` que define los pasos para construir una PC (`agregarCPU(), agregarRAM()`, etc.). Cada subclase concreta de Builder implementa estos pasos según una lógica distinta:

- `BuilderPCPersonalizada`: construye una PC a partir de las elecciones del usuario
- `BuilderPCEstudiante, BuilderPCGamer, BuilderPCPuntoVenta`: crean PCs prearmadas con configuraciones fijas

Finalmente, una clase directora `DirectorBuilder` centraliza el proceso y permite al cliente solicitar una PC sin preocuparse de como se ensambla internamente. Esta clase recibe los parámetros necesarios y delega la construcción al builder adecuado. Así, las clases en el paquete `modelo.builder` son las siguientes:

- `BuilderPC`: clase abstracta que define el plan de construcción
- `BuilderPCPersonalizada`: ensambla la PC en base a elecciones del usuario
- `BuilderPCEstudiante, BuilderPCGamer, BuilderPCPuntoVenta`: modelos predefinidos
- `DirectorBuilder`: clase directora que orquesta la construcción
- `PCBase`: producto final ensamblado (única clase en el paquete `modelo.decorador`).

Cada `BuilderPC` utiliza internamente el `DirectorFabrica` para obtener los componentes reales. Así, el patrón Builder se encarga del cómo se ensamblan los componentes, mientras que Abstract Factory se encarga del qué componentes se construyen.

- Decorator

El patrón Decorator nos permite añadir responsabilidades o funcionalidades adicionales a un objeto existente de forma dinámica y sin modificar su estructura interna. Este patrón es ideal cuando:

- Se quiere evitar una explosión de subclases para representar todas las combinaciones posibles
- Las funcionalidades adicionales son opcionales y pueden ser agregadas en distintas combinaciones

En nuestro caso, una PC ensamblada puede llevar software adicional, como Windows, Office, Photoshop, etc. Como estos elementos son opcionales y acumulables, Decorator nos permitió representarlos como capas que envuelven al objeto base sin alterar su lógica central.

De esta forma, la menera de aplicarlo a nuestro programa fue creando una interfaz `PC` con dos métodos clave: `obtenerCosto()` y `toString()`. La clase `PCBase` (la PC ensamblada con componentes de hardware) implementa esta interfaz. Luego, definimos una clase abstracta `DecoradorSoftware` que también implementa `PC` y contiene una referencia a otro objeto `PC`. Cada decorador concreto representa un software adicional: Windows10, Windows11, Office365, Photoshop, AutoCad, TerminalWSL. Cada uno agrega su propio costo y descripción, y puede apilarse con otros decoradores en cualquier orden. Entonces, las clases contenidas en el paquete `modelo.decorador`:

- `PC:` interfaz base:
- `PCBase:` componente concreto, representa la PC ensamblada.
- `DecoradorSoftware:` clase abstracta decoradora
- `Windows10, Office365, AutoCad`, etc. : decoradores concretos.

Con el uso de Decorator, pudimos extender de forma modular la funcionalidad del sistema sin necesidad de modificar `PCBase` o crear nuevas subclases por cada combinación de software. Esto permite agregar o eliminar software sin afectar la lógica del ensamblado.

- **Adapter**

El patrón Adapter se utiliza cuando tenemos clases incompatibles que deben trabajar juntas. Actúa como un "puente" entre dos interfaces. Este patrón es útil cuando:

- Se desea integrar clases existentes que no pueden ser modificadas.
- Se quiere estandarizar el uso de objetos que tienen diferentes estructuras o comportamientos

En este proyecto, el catálogo original de MonosChinos MX estaba centrado en componentes de Intel y NVIDIA. Sin embargo, la empresa decidió expandirse para incluir procesadores AMD, lo que origino un problema: estos procesadores no son directamente compatibles con el resto de los componentes del sistema (por ejemplo, las placas base).

Modificar todas las clases y el flujo del sistema para dar soporte explícito a AMD iba en contra de los principios de bajo acoplamiento. Por lo tanto, modelamos las CPU de modelo AMD de forma que fueran incompatibles con la interfaz CPU, luego usamos Adapter para integrar las CPUs AMD en el sistema como si fueran objetos del tipo CPU (Intel), permitiendo mantener la compatibilidad sin alterar el código principal.

Aplicando, creamos una interfaz `CPUAMD` que representa las CPUs de AMD con su propia estructura (atributos y métodos). Luego, definimos clases concretas como: `AMDRyzen5_5600G`, `AMDRyzen7_7700X`, etc.

Posteriormente, implementamos un adaptador llamado `CPUAMD_Adaptador`, el cual extiende la clase `CPU` (usada por el resto del sistema) pero internamente contiene una instancia de `CPUAMD`. El adaptador reimplementa los métodos de `CPU` redirigiéndolos hacia la CPU AMD interna. De este modo, cualquier clase que espera trabajar con una instancia de `CPU` (como `PCBase`) puede utilizar sin problema un `CPUAMD_Adaptador`. Asi, nuestras clases contenidas en el paquete `modelo.adapater`:

- `CPU:` clase abstracta del sistema.
- `CPUAMD:` interfaz AMD.
- `AMDRyzen5_5600G, AMDRyzen7_7700X`, etc.: implementaciones de `CPUAMD`
- `CPUAMD_Adaptador`: adaptador que extiende `CPU` y contiene una instancia de `CPUAMD`.

Por lo que gracias al uso del patrón Adapter, el sistema sigue trabajando solo con objetos del tipo CPU, y facilita la escalabilidad si en el futuro se desea integrar más marcas incompatibles.

- Singleton

El patrón Singleton asegura que una clase tenga una única instancia en todo el sistema y proporciona un punto de acceso global a ella. Es especialmente útil cuando:

- Se necesita un objeto centralizado que coordine o almacene información global
- Se requiere controlar el acceso concurrente desde distintos puntos del programa (como múltiples hilos)
- La creación de múltiples instancias causaría confusión o errores lógicos

En el contexto de este sistema, existe una entidad central que debe recibir y ensamblar todos los pedidos generados desde las distintas sucursales: la central CDMX. Esta entidad representa la única sede de ensamblaje y distribución, por lo que su representación en el código debía ser única y accesible globalmente. Creamos la clase `CentralCDMX` siguiendo el patrón Singleton:

- Se declaró una instancia estática y volatile para asegurar la visibilidad entre hilos
- Se implementó un método `getInstancia()` sincronizado con double-checked locking para asegurar la creación de una sola instancia incluso en entornos concurrentes
- Se hizo privado el constructor para evitar que otras clases puedan instanciarla directamente

Toda `PC` es ensamblada a través del método `CentralCDMX.nuevaPC(Pedido pedido)`, lo que garantiza que todas las solicitudes pasen por la misma instancia, ya sea desde clientes reales o simulados. Nuestra clases involucaradas:

- `CentralCDMX`: clase Singleton en el paquete `modelo.singleton`:
- `ControladorPC`: accede a la instancia para ensamblar la PC solicitada por el usuario

- `ClienteSimulado`: también accede a la misma instancia para enviar pedidos concurrentemente en el paquete `simulacion` (apartado del paquete `modelo` y de la MVC porque no es necesario para el funcionamiento principal del programa).
- `DirectorBuilder y DirectorFabrica`: son utilizados dentro de `CentralCDMX`, pero fueron diseñados como clases estáticas (no Singletons).

El uso de Singleton en `CentralCDMX` permitio asegurar que todas las operaciones de ensamblaje pasen por una única entidad y coordinar múltiples solicitudes concurrentes de forma segura y controlada. Esto también facilitó el manejo de pruebas con `ClienteSimulado`, ya que todos los hilos comparten y afectan la misma instancia, manteniendo la coherencia del sistema.

- **MVC**

El patrón MVC divide una aplicación en tres componentes principales:

- Modelo: la lógica del negocio y los datos.
- Vista: la interfaz de usuario (en este caso, consola)
- Controlador: el coordinador que conecta las entradas del usuario con el modelo y actualiza la vista

Este patrón es esencial cuando se desea emtre otras cosas, separar la presentación de la lógica del sistema, permitir la evolución de la interfaz sin afectar la lógica de negocio, y reutilizar la lógica del sistema con distintas interfaces.

Aplicamos MVC a lo largo de todo el flujo de interacción:

- Vista:
    - `VistaGeneral`: menú de selección de sucursal y consulta de pedidos.
    - `VistaSucursal:` interfaz por consola que guía al cliente paso a paso para elegir tipo de compra, componentes y software.
- Controlador:
    - `ControladorPC`: clase que dirige el flujo de compra dentro de una sucursal. Orquesta las llamadas a la vista, genera el Pedido, lo envía a CentralCDMX y muestra el ticket al usuario.
- Modelo:
    - `Pedido`, `Sucursal`, `SucursalJalisco, SucursalYucatan, SucursalChihuahua, ComponenteIncompatibleException,` etc.
    - Los paquetes `fabricaabstracta, adaptador, decorador, builder, singleton`.

Gracias al uso de MVC, podemos: separar la lógica de negocio y entrada/salida; reutilización del modelo desde distintos puntos (como `ClienteSimulado`).

Consideraciones y Comentarios

- El sistema incluye un cliente simulado que genera pedidos aleatorios en segundo plano, usando la clase `ClienteSimulado` implementada con `Runnable` y ejecutada en un hilo independiente. Permite probar el comportamiento concurrente del sistema mientras un usuario real interactúa con la interfaz.
- El sistema detecta hasta el ensamble de la PC en `CentralCDMX` si el cliente selecciona un procesador AMD incompatible con la placa base. Cuando esto ocurre, se lanza una excepción personalizada `ComponenteIncompatibleException`, y se realiza una adaptación automática usando el patrón Adapter. El usuario es informado de esta acción a través de un mensaje en consola.
- Cada pedido exitoso genera un ticket detallado que se almacena en un archivo de texto pedidos.txt. El sistema permite consultar todos los pedidos almacenados desde el menú principal.
- Hay algunas restricciones o limitaciones que se impusieron en el programa, sin violar los lineamientos del proyecto:
    - Todos los pedidos deben contener al menos un componente de cada tipo (CPU, RAM, GPU, etc.). No es posible ensamblar una PC incompleta.
    - En los componentes que lo permiten (RAM, HDD, SSD), se puede seleccionar entre 1 y 4 unidades.
    - Solo se puede agregar una vez cada tipo de software a la PC.
- Consideramos que el sistema fue completado en su totalidad. No hay funcionalidades faltantes con respecto a lo solicitado en la descripciónde la práctica.

Compilación y Ejecución

1. Ubicarse a la altura del directorio src.

2. Crear una carpeta para guardar los archivos .class:

```
mkdir -p bin
```

3. Busca recursivamente todos los archivos .java dentro de src, y luego los compila:

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

4. Ejecutar el programa:

```
java -cp bin MonosChinosMX
```