Instituto Tecnológico de Costa Rica

Bachillerato de Ingeniería en Computación

Grupo 1

Proyecto Programado 2

Patrones de Diseño

Estudiantes:

-Nayely Francini Chavarría Guevara (2021093564)

- Ian Steven Coto Soto (2021121435)

Prof. Yuen Law

Semestre I, 2023

Solución implementada

Para el desarrollo de este proyecto, luego del análisis meticulosos de las especificaciones proporcionadas, se optó por separar el proyecto en tres paquetes principales: el salón, la cocina y la simulación. Dada la necesidad de aplicar patrones de diseño y estructuración en el proyecto, se decide aplicar a los paquetes anteriormente mencionados el patrón estructural MVC (Model-View-Controler).

Módulo Salón

Para especificar un poco la separación de clases de cada paquete, iniciando en el salón se tiene por modelo, a la clase Salon; dado que en el salón hay un conjunto de mesas, se opta por crear una clase Mesa además para aplicar uno de los patrones de diseño aprendidos, se observa que es conveniente tener un patrón Factory para la creación de dichas mesas esto se logra a través de la clase FactroyMesas. En el View del salón se tiene la clase vistaSalon, con la cual se logra mostrar la parte gráfica del salón que consiste en un grupo de mesas, hechas por una matriz de botones; luego una sección donde el usuario pueda seleccionar el tipo de orden y los ingredientes extras en la orden además de mostrar en una tabla la información de todas las mesas ocupadas así como el estado de la orden de dicha mesa, también mostrará algunos botones que permitan agregar, enviar y pagar la orden. En el Controler que contiene a la clase controlerSalon ésta realiza una instancia del View y Modelo anteriores para manejar la comunicación de ambas partes así como realizar la conexión con la cocina.

Módulo Cocina

En el paquete cocina su Modelo corresponde a la clase ModeloCocina dado que en la especificación se menciona que la cocina recibe las órdenes del salón se es adecuado tener un Arraylist que almacene dichas órdenes y el número de mesa asociado a esa orden, para tener

menos complicaciones al momento de comunicar al salón qué orden está lista, el View de la cocina se consideró necesario que el usuario observará que órdenes están disponibles para entregar, se decide entonces tener una tabla con la información de la orden y la mesa además del estado de la orden, ahora bien ya que el usuario es quien decide qué orden completar, se optar por colocar un botón que acceda al número de la fila seleccionada de la tabla para que le indique al salón que esa orden está completada. En el Controler que contiene a la clase controladorCocina ésta realiza una instancia del View y Modelo anteriores para manejar la comunicación de ambas partes así como realizar la conexión de vuelta con el salón.

Módulo Simulación

En el módulo de Simulación también se aplica MVC. En el modelo se encuentra Simulación como pieza central que contiene una instancia de FactoryOrdenes cuyo objetivo es crear una orden cuando lo pida el usuario, que a su vez, tiene un FactoryHamburguesas que crea varias hamburguesas para la orden. Se cuenta con un booleano en el modelo para así, controlar el hilo que envía dados al Salón. También, con un ClienteSimulacion que se conecta con server (Salon) para mandar las órdenes generadas. En la vista (ViewSimulacion), solo se cuenta con dos botones, uno que sirve para enviar una orden manualmente, y otro que sirve para reanudar y pausar el hilo. Por último, ControlerSimualcion cuenta con dos atributos, una de vista y otro del modelo para poder obtener los datos y mostrarlos de buena manera y sin mezclar las funciones de una clase con las de la otra.

Paquetes generales

Otros aspectos a considerar fueron la orden y el tipo de producto que se ofrecía, por lo que se opta por crear un paquete para cada aspecto, uno llamado Orden que contiene la clase orden la cual implementa serializable, dado que el usuario puede ingresar una cantidad desconocida de productos en una sola orden, se opta por tener un Arraylist de Hamburguesas,

que es el único producto ofrecido así como un atributo que permita indicar el estado de la orden; en esta clase también se toma en cuenta que debe poseer métodos que permitan ingresar la hamburguesa, calcular el costo y obtener el costo total de la orden; en cuanto al tipo de producto corresponde a una hamburguesa básica, por lo que se crea un paquete llamado Comida, este contiene en una carpeta llamada Hamburguesa un interface de Hamburguesa que implementa serializable y se implementa en la clase HamburguesaBasica, hay una carpeta que contiene los tipos de ingredientes extras, que utilizan la clase IngredienteDecorador está clase permite darle dinámicamente el comportamiento de una hamburguesa a eso ingredientes, por eso dicha clase implementa el interface hamburguesa.

Conexiones

Tomando en consideración que se deben comunicar el salón con la cocina y viceversa así como la simulación con el salón, se opta por usar el patrón estructural Cliente-Servidor; para ello se crea un paquete que contiene las clases server, para realizar la conexión salón-simulación, ServerCocina que conecta al salón-cocina, es decir recibe las órdenes provenientes del salón; ServerSalon que recibe las órdenes completadas por la cocina mientra que para el cliente se tiene una única clase ClienteCocina esta sirve para hacer la conexión ida y vuelta del salón-cocina, para esta forma de comunicación se consideró que no era posible usar un único puerto, por ende, las clases que sirven de servidores tiene puertos diferentes, en cuanto al cliente se optó por tener un atributo int para indicar el número de puerto y uno que almacene el mensaje que se debe enviar los cuales se reciben en el constructor de dicha clase.

Explicada la forma de comunicación, es necesario establecer qué objeto es el que se enviará a través de las conexiones, por lo que se optó enviar un arreglo de tipo Object conformado por entero correspondiente al número de mesa y una orden para enviar en la comunicación salón-cocina mientras que en la conexión cocina-salón el arreglo solo es formado por el número de mesa. En cuanto a la conexión simulación-salón se optó por enviar una Orden, por lo que el servidor encargado de esta comunicación tiene un atributo de ese tipo.

Modelado de clases

En esta sección se dará una breve explicación sobre las clases de los diversos paquetes utilizados para dar solución al proyecto además del diagrama de clases asociado a la implementación de este y los diagramas que muestran la interacción de cada clase. Además, también se muestran los diagramas de actividad.

Paquete de clases Comida

Este paquete cuenta con dos subcarpetas que permiten la creación de las órdenes que se generan en el salón como en la simulación. Entre ellas están la subcarpeta Hamburguesa que es la parte principal de la orden ya que es el único tipo de producto ofrecido e Ingredientes que está conformada por todos los ingredientes extras que puede llevar la hamburguesa. Además, cuenta con un interface con los precios de los ingredientes extras disponibles.

Subcarpeta Hamburguesa

Aquí se encuentran el interface Hamburguesa que únicamente contiene los métodos getDescripcion() este retorna un string con la descripción de la hamburguesa y getPrecio() que retorna un double, esta clase implementa serializable. Luego se encuentra la clase HamburguesaBase que implementa Hamburguesa solo cuenta con los mismos atributos de esa clase.

Subcarpeta Ingredientes

Esta subcarpeta tiene varias clases que representan los diversos ingredientes extras, entre ellos el queso, lechuga, mayonesa, tomate, jalapeño, tocina, ketchup, huevo frito, pepinillos, aguacate y cebollas esta clases heredan de la clase IngredienteDecorador, que implementa el interface Hamburguesa, está clase facilita el hecho de no tener que crear

muchas clase de hamburguesa; ya que este decorador le permite a esos ingredientes tomar el comportamiento de la hamburguesa.

Paquete de clases Orden

La clase correspondiente a este paquete es la clase Orden, que implementa serializable debido a que este objeto debe enviarse a través de sockets, esta cuenta con un Arraylist de hamburguesas, un booleano que indica el estado de la orden y un double para indicar el precio total de la orden. En su constructor se inicializa el Arraylist y se establece el estado de la orden en false.

Entre otros métodos se tiene los siguientes getters para obtener el Arraylist de las hamburguesas, el precio y el estado de la orden, por setters se tiene uno para obtener el estado de la orden, hay también dos métodos que retorna un string con la información de la orden, uno incluye el estado de la orden y no lo incluye; los métodos principales de esta clase son agregarHamburguesa(Hamburguesa newHamburguesa) este agrega la hamburguesa a el Arraylist y además llama al método que determina el costo de esta misma hamburguesa que es agregarCosto(Hamburguesa newHamburguesa).

Paquete de clases Simulación

Este módulo se divide en tres subcarpetas que separan los archivos de la vista, controlador y modelo, aplicando el MVC.

Subcarpeta Model

Esta carpeta cuenta con todos los archivos relacionados al manejo de datos. Consta de 3 clases y un interface: FactoryHamburguesas, FactoryOrden, Simulación y Constantes. FactoryHamburguesas crea una hamburguesa de forma aleatoria, cuenta con ingredientes random (se usa una constante de Constantes para así, poder tener la cantidad máxima de

ingredientes que existen) y se actualiza su precio en base a eso. FactoryOrden crea una orden con una cantidad de hamburguesas máxima definida en Constantes; utiliza un atributo de FactoryHamburguesas para obtener cada hamburguesa de manera aleatoria y que no se repitan, estas son agregas a la orden que tiene un ArrayList<Hamburguesa>. Por último, Simulacion que tiene un FactoryOrden para poder generar las órdenes, un booleano que representa si el hilo está activo o no, y un cliente (ClienteSimulacion) que envía la órden al servidor de salon (server).

Subcarpeta View

En la vista solo se cuenta con una clase, ViewSimulacion. La vista contiene dos botones, uno encargado de enviar una orden al Salon, y otro para pausar o reanudar el hilo. La acción al presionar el botón se le da en la clase Controler.

Subcarpeta Controler

En el modelo se muestran dos clases: Constantes (sirve para tener el tiempo de espera entre órdenes generadas por el hilo) y ControlerSimulacion. La última, es la encargada de todas las interacciones con el usuario y comunicar el modelo con la vista. Para esto, se crean dos atributos, uno del modelo y uno del salon. Se añade el addActionListener de los botones aquí y sus respectivas funciones: enviarOrden() y cambiarEstadoHilo(); cada una para el botón que cumple esa función. Al presionar el botón de “Enviar orden”, se llama a la función que hace que el modelo genere una nueva orden, la pase al Salon por medio del cliente, por último, genere una notificación en la vista retroalimentando al usuario con un “Orden enviada a salón”. Al presionar el botón relacionado al hilo, llamada al método cambiarEstadoHilo() que toma el booleano contrario del modelo (si es true, se toma false), se lo asigna, por último, pasa a la vista el booleano y cambiará el texto del botón y saldrá una notificación correspondiente a la acción realizada.

Paquete de clases Salón

Este paquete se divide en tres subcarpetas que conforman toda la lógica respectiva para dar solución al módulo Salón.

Subcarpeta ControlerSalon

En esta subcarpeta se encuentra la clase controlerSalon la cual se encargará de unir al modelo y la vista respectivas del salón, por ende, se cuenta con un atributo de esos dos elementos, además habrá un atributo Pattern donde se tendrá una expresión regular para buscar dígitos dentro de un String. En lo referente a los métodos de esta clase se encuentra el constructor, el cual inicializa el modelo y la vista, además creará los ActionListener de los botón para pagar, los botones que representan las mesas y el botón para realizar el envío del pedido a la cocina, que se encuentran en la vista del salón.

Otros métodos de esta clase son pagarPedido(JButton btn), correspondiente al botón a pagar, en este se obtiene la fila seleccionada de la tabla informativa en la vista; sino se selecciona una fila muestra un mensaje en pantalla, una vez obtenido el valor numérico correspondiente a la primera columna de la fila se obtiene la posición de la mesa dentro de la matriz de mesas en el modelo, luego realizan algunas validaciones con respecto al estado de la orden, se muestra un mensaje si la orden no está lista y en casa de estar lista se procede a obtener la factura de dicha mesa para mostrar la información al usuario.

Con el método clickMesa(JButton btn), se muestra la información de la mesa correspondiente al botón que se ha presionado en la vista. En el método ObtenerMesa(int id_mesa), con este método lo que se busca es obtener la posición de la mesa con ese número dentro de la matriz de mesas del salón, de modo que accede a la matriz a través del modelo, por lo que retorna un arreglo de enteros.

Finalmente se tiene el método enviarPedido(JButton btn), se obtiene la orden de la vista, se hace una validación para asegurarse que el usuario ha creado una orden sino lo ha

hecho muestra un mensaje luego se procede a buscar una mesa libre de no haber se informa al usuario, en caso de haber una mesa disponible se agrega la orden a la mesa, se muestra la información de la mesa en la tabla informativa y se procede a enviar la orden a la cocina, a través del cliente conectado al servidor de la cocina.

Subcarpeta ModelSalon

En esta subcarpeta hay tres clases; la primera de ellas es un FactoryMesas con esta se crea la matriz de mesas la cual se obtienen con el método getMesas(), por ende, se tiene un atributo público que indica el tamaño de dicha matriz.

Luego se tiene la clase Mesas, se tienen atributos para indicar el estado de la mesa (booleano), un identificador (int), fila y columna (ambas int), una orden (Orden). Con respecto al constructor de la clase se recibe el identificador, fila y columna que se asignan respectivamente además se establece el estado de la mesa con false.

Otros métodos de la clase se tienen infoPedido() que retorna un String con la información de la orden, pagarCuenta() retorna un double equivalente al total de la cuenta de la orden además establece el estado de la mesa en false indicando que está libre, cambiarEstadoOrden() este simplemente pone el estado de la orden en true, getInfo() este retorna un String con toda la información de la mesa, getId_mesa() este retorna el valor del identificador, agregarOrden(Orden orden) este agrega la orden a la mesa, getOrden() retorna la orden de la mesa, isEstado() retorna el valor del estado de la mesa, setEstado(boolean estado) con este se cambia el estado de la mesa, getPosicion() retorna un int[] con los valores de la fila y columna.

Finalmente se tiene la clase salon, solo cuenta con un atributo para la matriz de las mesas, en el constructor se llama al FactoryMesas para asignarlo al atributo del salón. Entre otros métodos están agregarOrden(int fila, int columna, Orden orden) este agrega la orden a la mesa que se encuentra en la fila y columna recibidas, obtenerCuenta(int fila, int columna) este

método retorna la llama al método pagarCuenta de la mesa en esa posición, obtenerInfoMesa(int fila, int columna) retorna la llama al método getInfo() esa mesa, obtenerInfoOrden(int fila, int columna) retorna un String con conformado por la llamada del método getId_mesa() y la llama al método getOrden().obtenerOrden() de esa mesa, obtenerMesaLibre() este busca dentro de la matriz de mesas retornando la primera mesa que tenga el estado en false, estadoMesa(int fila, int columna) retorna la llamada al método isEstado() de la mesa en esa posición de la matriz, cambiarEstadOrden(int id) a partir del identificador recibido al encontrar la mesa con el mismo identificador llama al método cambiarEstadOrden() de la clase Mesa, obtenerMesa(int id_mesa) este recorre la matriz y retorna la posición de la mesa con ese identificador en caso de no encontrar retorna nulo, estadoOrden(int fila, int col) este retorna las llamadas getOrden().getListo() de la mesa en la posición recibida.

Subcarpeta ViewSalon

Esta subcarpeta cuenta solamente con la clase vistaSalon, como atributos públicos se tiene una matriz de JButton (mesas), los JButton enviarOrden y factura, model (DefaultTableModel) y el JTable facturar; ya que estos son accedidos por el controler. El resto de atributos son privados y conforman los ckeckbox con las opciones de ingredientes extras, los paneles para agregar todos los componentes, también algunos labels que contienen información.

Por métodos se tiene agregarComponentes() y confiPnlDerecho() que se encargan de configurar los elementos de la ventana, agregarMesas() se encarga de generar los botones que representan las mesas del salón, agregarHamburguesa(JButton btn); este método está asociado al botón agregarOrden, se encarga de agregar la hamburguesa a el atributo orden, posicionMesa(JButton btn) se retorna un int[] de la posición del botón recibido en la matriz de botones y por último está stateChanged(ChangeEvent e) que se asocia a los ckeckbox y un

RadioButton con este se crean los diversos objetos de los ingredientes extras y la hamburguesa básica.

Paquete de clases Cocina

Este paquete se divide en tres subcarpetas que conforman toda la lógica respectiva para dar solución al módulo Cocina.

Subcarpeta ControlerCocina

Esta subcarpeta cuenta con la clase ControladorCocina que tiene por atributos un ModeloCocina (cocina) y View (vistaCocina) ambos son públicos, en la parte del constructor se inicializan estos atributos además de acceder al botón completarOrden del atributo vistaCocina, con el fin de inicializar el actionListener de ese botón. Por último, está el método enviarPedido(JButton btn), correspondiente al actionPerformed del botón completarOrden, este simplemente accede a la fila seleccionada de la tabla informativa en la vista; una vez confirmada la selección elimina la fila de la tabla y envía, a través de una instancia de ClienteCocina, un Object[] con el número de la orden completada; para luego eliminar esa orden de la cocina.

Subcarpeta ModelCocina

Esta carpeta contiene a la clase ModeloCocina que tiene un único atributo, Arraylist<Object[]> órdenes; en el constructor se inicializa este atributo. Se crean tres métodos públicos uno para eliminar una orden del Arraylist, otro que busca una mesa en específico y retorna el número de la mesa; ambos métodos reciben un entero y otro para agregar la orden el cual recibe un object[].

Subcarpeta ViewCocina

La clase View, conforma esta subcarpeta, se tiene por atributos un Jtable(ordenes) y DefaultTableModel (model) además de un Jbutton (completarOrden) estos atributos son públicos dada la necesidad de accederlos por medio del controlador, por atributos privados se tienen al JFrame y un JscrollPane. En cuanto a los métodos se tienen iniciarComponentes(), el cual se llama a través del constructor y básicamente se encarga de crear y configurar todos los componentes gráficos de la ventana.

Paquete de clases ServerClient

Este paquete se utiliza para almacenar las clases encargadas de comunicar los módulos de la cocina y el salón, mediante el uso de sockets. Entre las clases que se encuentran ServerCocina, que es un servidor destinado para la recepción de órdenes enviadas por el salón a través del puerto 1234; entre sus atributos se encuentran un object[], ObjectInputStream, ControladorCocina además de un serverSocket y un socket; en su constructor se recibe el ControladorCocina además se configura el hilo que manejara la ejecución del servidor, ServerSalon es parecida a la clase ServerCocina solo que trabaja en un puerto diferente (puerto 5555) además de que recibiría las órdenes listas que envía la cocina además tiene los mismo atributos de esa clase con la única diferencia de que en lugar del ControladorCocina hay un controlerSalon, el constructor hace lo mismo solo que recibe un controlerSalon. Otro de los métodos de estas clases es el método run() asociado al hilo de cada clase, con este se logra mantener al servidor en espera de un cliente, una vez conectado el cliente el método lee el mensaje enviado por el cliente y lo asigna al object[]; en el caso del ServerSalon el método debe cambiar el estado de la orden que la cocina le envió tanto en el modelo del salón como en la vista, algo parecido sucede con ServerCocina, este luego de leer el mensaje agrega la orden al modelo de la cocina y refleja esa información en la vista.

Otro clase que sirve de servidor es server, esta funciona exclusivamente para establecer la conexión desde la simulación hasta el salón a través del puerto 9999; en cuanto a sus atributos se tiene un salon, una orden, ObjectInputStream, un serverSocket y un socket; para el constructor de la clase se recibe un salón el cual se asigna al atributo con el mismo nombre además se crea un hilo para mantener el servidor en ejecución, entre otros métodos de la clase se tiene buscarMesa() que obtiene del salón una mesa libre, también está el método run() asociado al hilo en este se mantiene al server esperando conexión, luego de haber establecido la conexión lee el mensaje del cliente y lo convierte en una Orden para luego asignarlo al atributo de este tipo, posteriormente revisa si el salón tiene mesas libres, en caso de no haber muestra un mensaje al usuario, sino entonces procede a asignar esa orden a la mesa libre que obtuvo del método buscarMesa().

La clase que se conecta con la anterior en ClienteSimulacion. Se conectan al mismo puerto (9999) y tiene dos atributos socket (cliente) y ObjectOutputStream (output). Lo que se hace es llamar a la función conectar(Orden newOrden), la misma recibe la orden, abre la conexión y envía el dato. Server debería recibirlo si está conectado. Ambos comparten el mismo tipo de dato, Orden.

Finalmente, la clase ClienteCocina, entre sus atributos se encuentran un socket (cliente), ObjectOutputStream (output), int (puerto) y un object[] (mensaje) esta se usa tanto para establecer la conexión del salón a la cocina como viceversa, esto es posible ya que en el constructor se recibe el número de puerto del servidor además del object[] que sería el mensaje, el otro método se encarga de realizar la conexión con el servidor, por lo que se inicializan el cliente con el número de puerto recibido y el output con el mensaje para luego enviar dicho mensaje.

Aplicación de Conceptos

Para este proyecto se aplicaron conocimientos aprendidos anteriormente como la herencia y el polimorfismo, así como nuevos conceptos entre ellos los sockets e hilos además de los patrones de diseño como Factory, observe, etc. y patrones estructurales como MVC, Cliente-Servidor y Publisher-Subscriber, en las siguientes secciones se explicarán las ventajas y desventajas de estos conceptos así como las áreas donde se aplicaron.

Aplicación de patrones

En este proyecto se aplicó cuatro patrones el patrón Decorator, Factory, MVC y Cliente-Servidor; a continuación se explicara el uso que se le dio a dichos patrones y las ventajas que trajeron al aplicarlos.

Patrón Decorator

El patrón Decorator se optó por aplicarlo en la parte de productos en este caso con el fin de proporcionar flexibilidad y extensibilidad en la creación de hamburguesas personalizadas con ingredientes adicionales. El patrón Decorator según la definición “permite agregar dinámicamente nuevas funcionalidades o características a un objeto existente sin modificar su estructura base”(*Decorator En Java / Patrones De Diseño*, s.f.). Es perfecto, ya que en lugar de crear múltiples clases de hamburguesas con diferentes combinaciones de ingredientes, se utiliza el patrón Decorator para agregar ingredientes adicionales de forma modular y escalable. Al implementar el interface de Hamburguesa y utilizar el patrón Decorator, se encadena y apila los ingredientes adicionales de forma flexible. Cada ingrediente adicional se convierte en un "decorador" que envuelve la hamburguesa base y proporciona una funcionalidad adicional. Por lo que al llamar a los métodos de descripción y obtener precio, el objeto decorado devuelve la descripción y precio actualizados según los ingredientes añadidos.

Patrón Factory

Se utilizó el patrón Factory en dos áreas del proyecto la primera se da en el módulo del salón esto con el fin de generar la mesas; la otra sucede en la simulación con la creación de las órdenes y hamburguesas. Este patrón permite abstraer la lógica de creación de las mesas y órdenes sin tener que agregar el código para generar estos objetos en diversas partes del proyecto lo que facilitó centralizar y gestionar de manera más eficiente la creación de mesas y órdenes en un único lugar. Este también permitió la creación de objetos de manera flexible y extensible. Ya que para las órdenes se podía definir diferentes tipos de ingredientes en la hamburguesa de esa orden en la fábrica y elegir cuál crear en función de ciertos parámetros o condiciones.

Patrón MVC

En el caso del patrón MVC(Modelo-Vista-Controlador) se utilizó en dos módulos el salón y la cocina; esto permitió separar claramente las responsabilidades de cada componente del sistema. Siendo el modelo el encargado de representar y manejar los datos relacionados con el salón o la cocina, la vista se encarga de la presentación de la interfaz de usuario y el controlador se encarga de coordinar las interacciones entre el modelo y la vista. Esta separación permitió mantener un código organizado, modular y fácil de mantener. Al tener separado el modelo de la vista y el controlador, cada componente puede evolucionar de forma independiente, facilitando la reutilización de los modelos de datos en diferentes vistas o incluso en otras partes del sistema sin afectar su funcionalidad subyacente, esto también facilitó la prueba unitaria y la prueba de integración, esto facilitó probar cada componente de manera independiente, lo que mejora la calidad del código y permite una detección temprana de posibles problemas. También permite que a futuro si es necesario agregar nuevas modificaciones a alguno de los modelos los demás no se verán afectados.

Patrón Cliente-Servidor

El patrón Cliente-Servidor es adecuado cuando se necesita una arquitectura distribuida, donde diferentes componentes o módulos se ejecutan en diferentes dispositivos o servidores. En este proyecto, el módulo de cocina y el módulo de salón cuentan con servidores separados, ya que ambos módulos deben comunicarse en ambos sentidos. Otra ventaja es la clara separación de responsabilidades entre el cliente y el servidor, en el caso del servidor de la cocina este tiene que recibir del salón las órdenes que se deben completar y el servidor del salón recibe esas órdenes completadas; mientras que los clientes solo enviarán los datos a estos servidores, el motivo de tener servidores en cada módulo se debe a que de esa forma no habría interferencia en el envío de mensajes además de que el salón también tiene que recibir las órdenes provenientes de la simulación, también facilita el hecho de realizar cambios o mejoras en el servidor sin afectar a los clientes.

Ventajas del uso de patrones

Decorator

La ventaja principal de este patrón es su versatilidad dadas las capas. Se pueden crear objetos base y modificar sus valores con el decorador necesario. Por ejemplo, cada ingrediente tiene un precio y al implementarlo en las interface y clases de decorator, se puede conseguir la suma de este adicional en el precio final.

Factory

Lo mejor de este patrón sin duda es la limpieza del código y la fácil obtención de instancias mediante una sola clase. Al tener cada factory necesario en una clase, se puede llamar y obtener su instancia sin necesidad de complicar mucho el código. Además, se evita la repetición de código si se necesitase obtener instancias en otras partes del código.

MVC (Model-View-Controler)

El MVC hace que el código quede más ordenado y legible. Al estar cada función separada, se puede ir al lugar al que ocurre un problema en caso de ocuparlo. También, simplifica la lógica, debido a que solo hay que pensar en los datos y la vista en cada paquete. Otra ventaja es que, si necesitara cambiar la vista para poder utilizarse en otros dispositivos como lo podrían ser smartphones, se podría hacer, sin tener que tocar la parte de los datos. Esa es su mayor ventaja.

Cliente-Servidor

El patrón Cliente-Servidor permite escalar tanto el cliente como el servidor de forma independiente. Además, facilita la modularidad en el diseño ya que el servidor se encarga de la lógica de negocio y la gestión de datos, mientras que el cliente se enfoca en la interfaz de usuario y la presentación de los datos. Además de ser bastante fácil de aplicar y comprender.

Desventajas del uso de patrones

Decorator

La gran desventaja de Decorator es la complejidad, debido a que es confuso al principio visualizar cómo funciona. Además de que los decoradores no se pueden eliminar en un futuro de la manera en la que lo implementamos, y hacerlo, requiere más complejidad.

Factory

No se observó ninguna desventaja del patrón. Cumple con su objetivo que es generar una instancia con decoradores aleatorios de gran manera.

MVC (Model-View-Controler)

Lo único malo con lo que cuenta el MVC es la complejidad para los novatos en este patrón, debido a que la separación de cada responsabilidad puede ser confuso para algunos. Además de entender realmente cómo funciona controler; qué debe de realizar.

Cliente-Servidor

Para este patrón tampoco se observó alguna desventaja, ya que es un patrón bastante sencillo de comprender y aplicar .

Referencias

Banas, D. (2011). *Decorator Design Pattern* [Video]. YouTube. https://www.youtube.com/watch?v=j40kRwSm4VE&list=PLiyhZuxw8t8Yz0cTUgVweaQeJZHP3raWA&index=2&ab_channel=DerekBanas

BettaTech. (2019). *DECORATOR | Patrones de diseño* [Video]. YouTube. https://www.youtube.com/watch?v=nLy4x_LPPWU&list=PLiyhZuxw8t8Yz0cTUgVweaQeJZHP3raWA&index=3&t=7s&ab_channel=BettaTech

BTech Smart Class. (s. f.). *Variables in Java Interfaces.* http://www.btechsmartclass.com/java/java-variables-in-interfaces.html

Códigos de Programación - MR. (2016). 4: MVC en Java (Modelo, Vista, Controlador) [Video]. YouTube. https://www.youtube.com/watch?v=GGOwrMKBKeY&t=922s&ab_channel=C%C3%B3digosdeProgramaci%C3%B3n-MR

IONOS. (19 de febrero de 2021). *Decorator pattern: explanation, UML presentation, and example.* https://www.ionos.com/digitalguide/websites/web-development/what-is-the-decorator-pattern/

Redkar, A. (2018). *What is MVC architecture?* [Video]. YouTube. https://www.youtube.com/watch?v=mtZdybMV4Bw&ab_channel=AbhayRedkar

The Coder Cave esp. (2019). *¿Qué es MVC? - Aprende MVC en 10 minutos!* [Video]. YouTube. https://www.youtube.com/watch?v=UU8AKk8Slqg&ab_channel=TheCoderCaveesp

Web Dev Simplified. (2018). *MVC Explained in 4 Minutes* [Video]. YouTube. https://www.youtube.com/watch?v=DUg2SWWK18I&ab_channel=WebDevSimplifie d

Portillo, G. (2023). *JTable: Agregar Filas*[Video]. YouTube.

https://www.youtube.com/watch?v=pQomQ1Op-nE&list=WL&index=1&t=357s

Portillo, G. (2020). *JTable: Modificar fila*.[Video] YouTube.

https://www.youtube.com/watch?v=lX9TcUzJ3B4&t=514s