**Taller 5 – patrones**

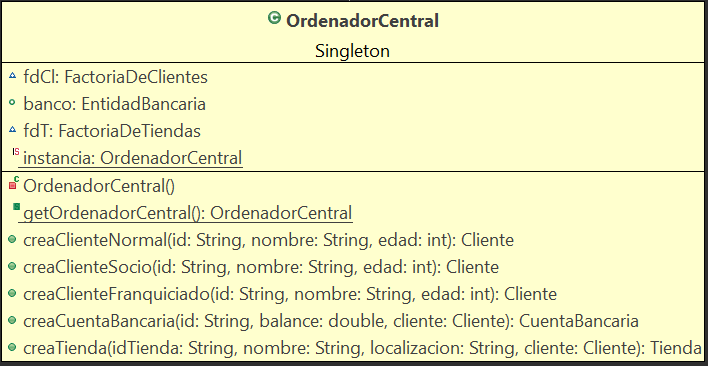
**Enlace repositorio guía:** https://github.com/AdrianGilGamboa/ProyectoPatrones

1. **Información general del proyecto**

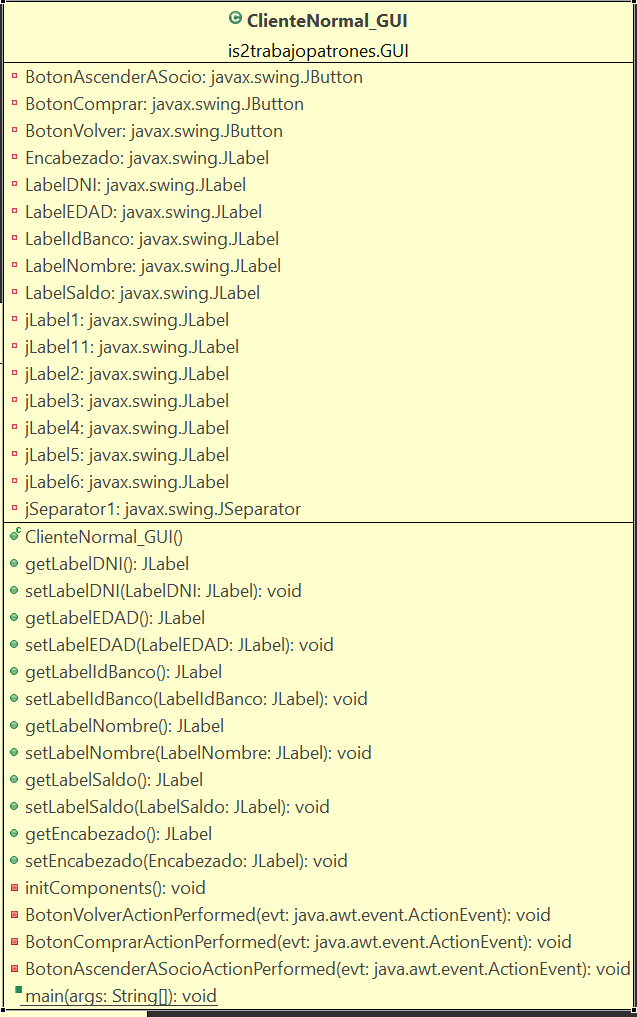
El proyecto tiene por objetivo la simulación del funcionamiento de una multinacional bancaria. De esta forma, en la aplicación es posible tener múltiples registros (cliente normal, socio o franquiciado). Así pues, sin importar la selección, el sistema proporcionará la opción de seleccionar un saldo que, en este caso hace referencia a la labor impartida por la multinacional. Adicionalmente, dependiendo del rol, dependerán muchas de las acciones que se van a poder realizar. Por ejemplo, si se es un cliente normal o socio, se va a tener acceso a las diversas tiendas que, en primera instancia han de ser creadas haciendo llamado en la clase “Multinacional”, la cual luego de ejecutar sus métodos, se terminará llamando a la clase “FactoriaDeTiendas”, de esta manera los clientes podrán comprar los productos que maneje cada una. Además, de la misma manera en que se crea una tienda, se evidencia un funcionamiento similar para la creación de clientes de cualquier tipo y su respectiva cuenta bancaria, siendo las encargadas de esta labor las clases “FactoriaDeClientes” y “FactoriaDeCuentas” en ese orden. Varias de las demás clases relacionadas con una cuenta, un cliente o una tienda, son especializaciones del tipo de cada una de ellas, como lo es el caso de los clientes: socios o normales. Por otra parte, muchas de las clases restantes se especializan en la creación de la interfaz y los llamados a las acciones que ejecutan los diferentes botones, como puede ser la opción de agregar productos en caso de ser una tienda o comprar un producto si se trata de un tipo de cliente. Finalmente, es fundamental mencionar la labor de la clase “OrdenadorCentral”, ya que se encuentra directamente implicada en cualquier acción relacionada con la creación de un objeto d tipo cliente, cuenta o tienda, logrando así la creación de un estado centralizado y para poder coordinar las operaciones del sistema de forma asertiva.

**Componentes claves del proyecto:**

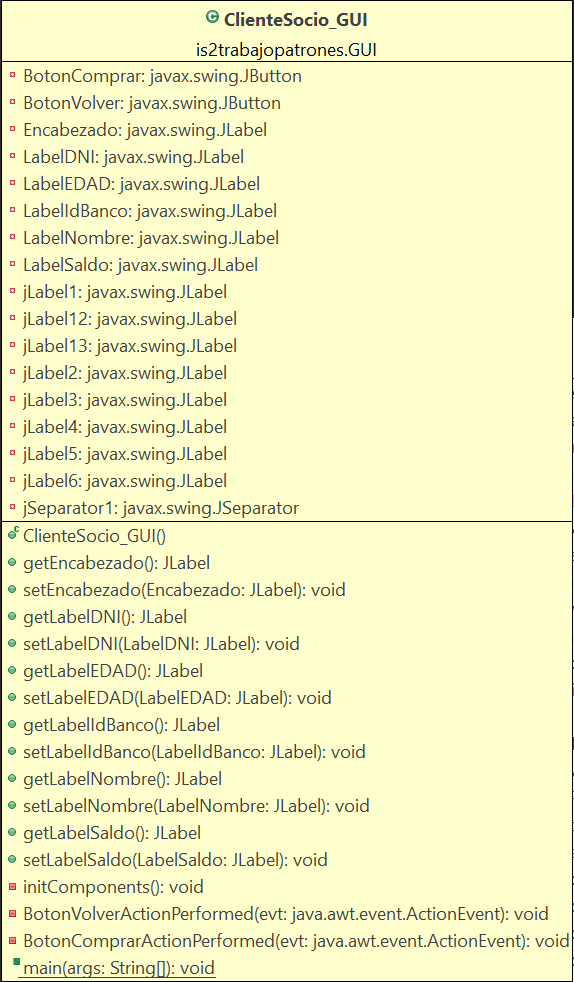
**Clase OrdenadorCentral:** Es el núcleo del sistema y actúa como un punto centralizado para la gestión de clientes, tiendas y cuentas bancarias. Esta clase implementa el patrón Singleton y proporciona métodos para crear diferentes tipos de clientes, cuentas bancarias y tiendas.



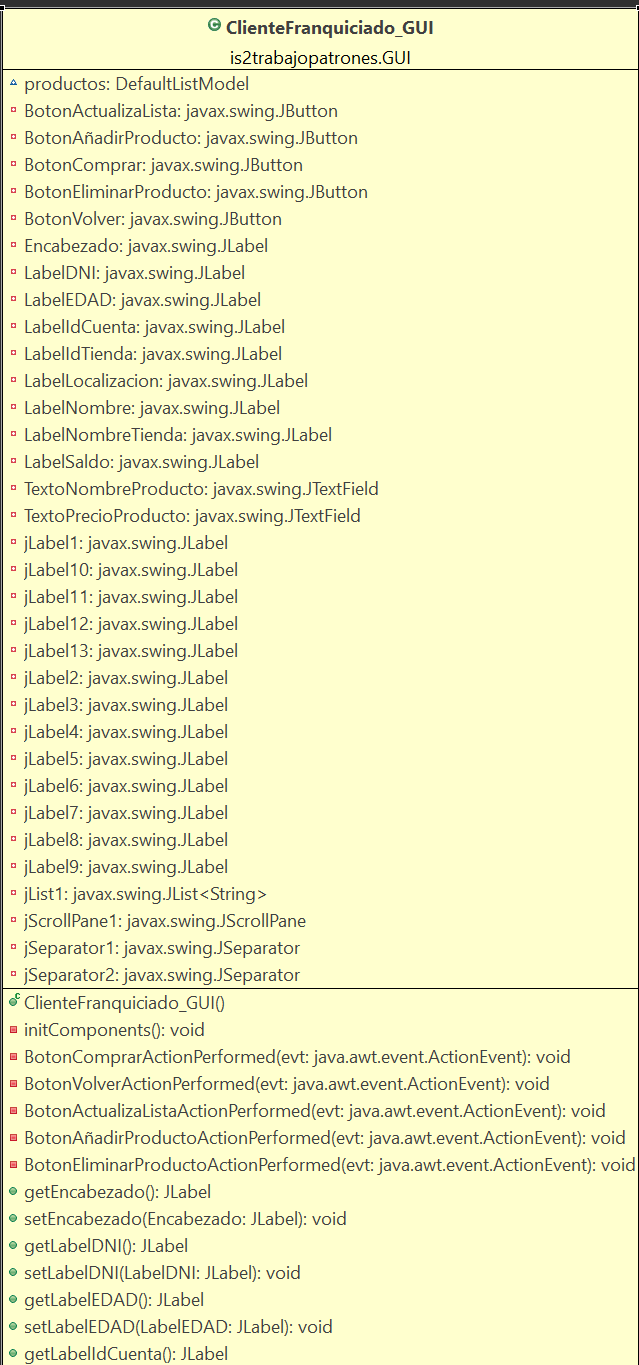
**Clases relacionadas con los clientes:** En el diseño del proyecto, existen clases y componentes relacionados con la gestión de clientes, como Cliente, ClienteFranquiciadoImpl, ClienteSocio, FactoriaDeClientes y estrategias asociadas. Estas clases permiten la creación y gestión de diferentes tipos de clientes y sus características específicas.



**Cliente Normal**

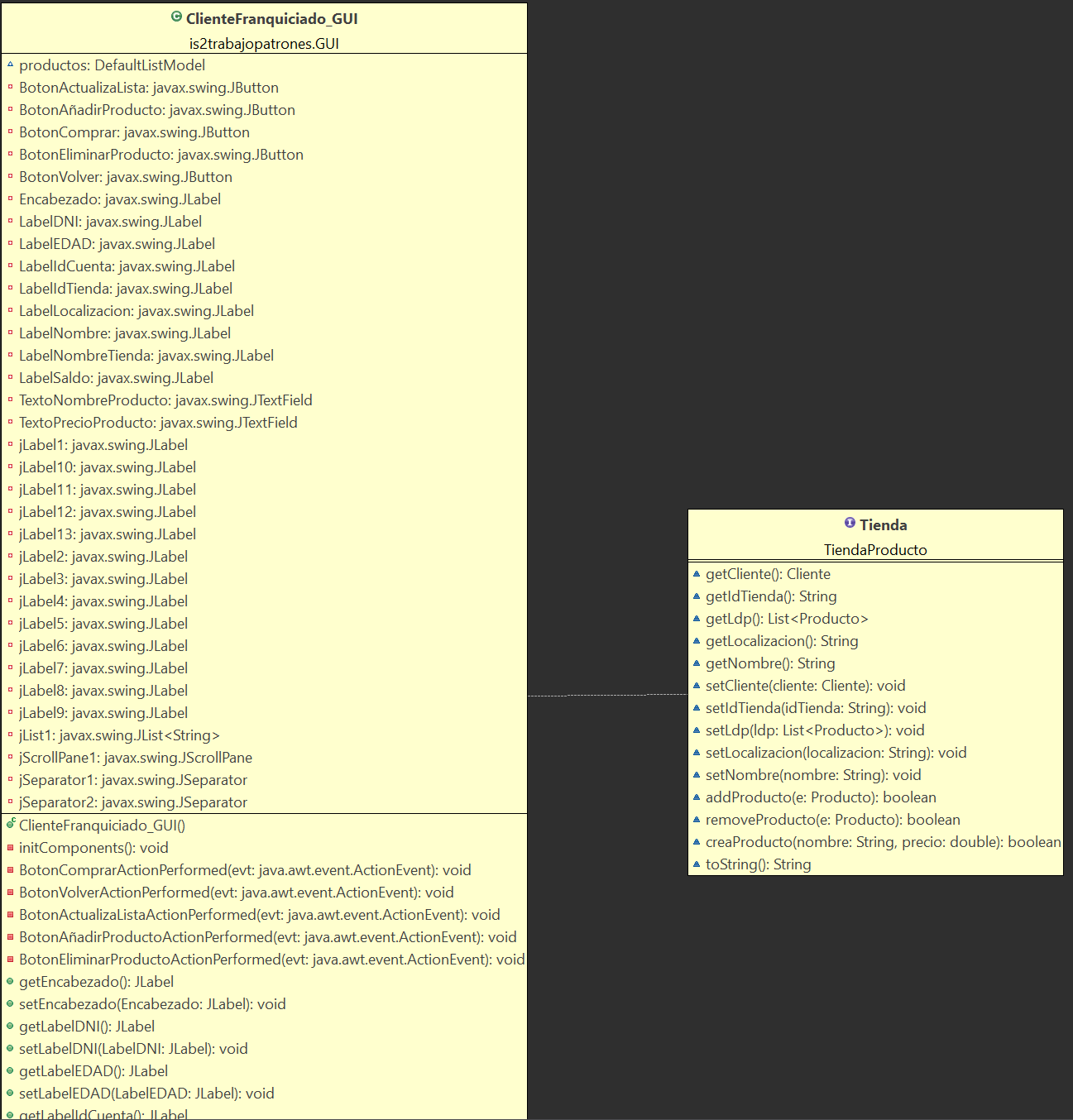
****

**Cliente Socio**

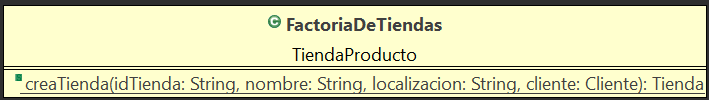
****

**Cliente Franquiciado**

**Clases relacionadas con las tiendas:** También hay componentes y clases relacionados con la gestión de tiendas, como Tienda, FactoriaDeTiendas y la asignación de tiendas a clientes franquiciados. Estas clases permiten crear tiendas, asignar clientes franquiciados a tiendas específicas y realizar operaciones relacionadas con las tiendas.

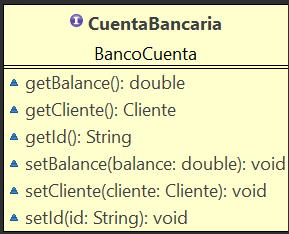


**Relación clase Tienda con ClienteFranquiciado**

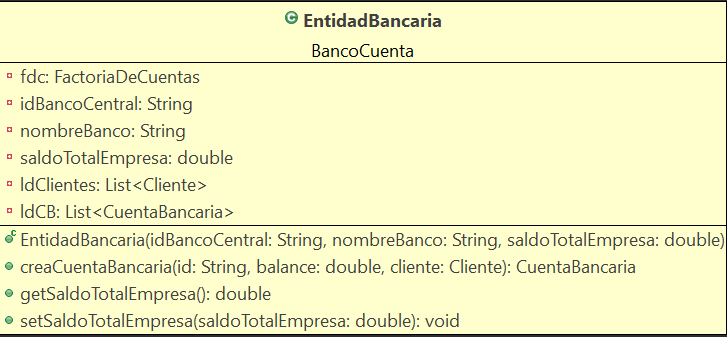
****

**Clase FactoriDeTiendas**

**Clases relacionadas con las cuentas bancarias:** El proyecto incluye clases como CuentaBancaria y EntidadBancaria para la gestión de cuentas bancarias. Estas clases permiten crear cuentas bancarias, realizar operaciones de depósito y retiro, y calcular saldos totales.



**Clase CuentaBancaria**

****

**Clase EntidadaBancaria**

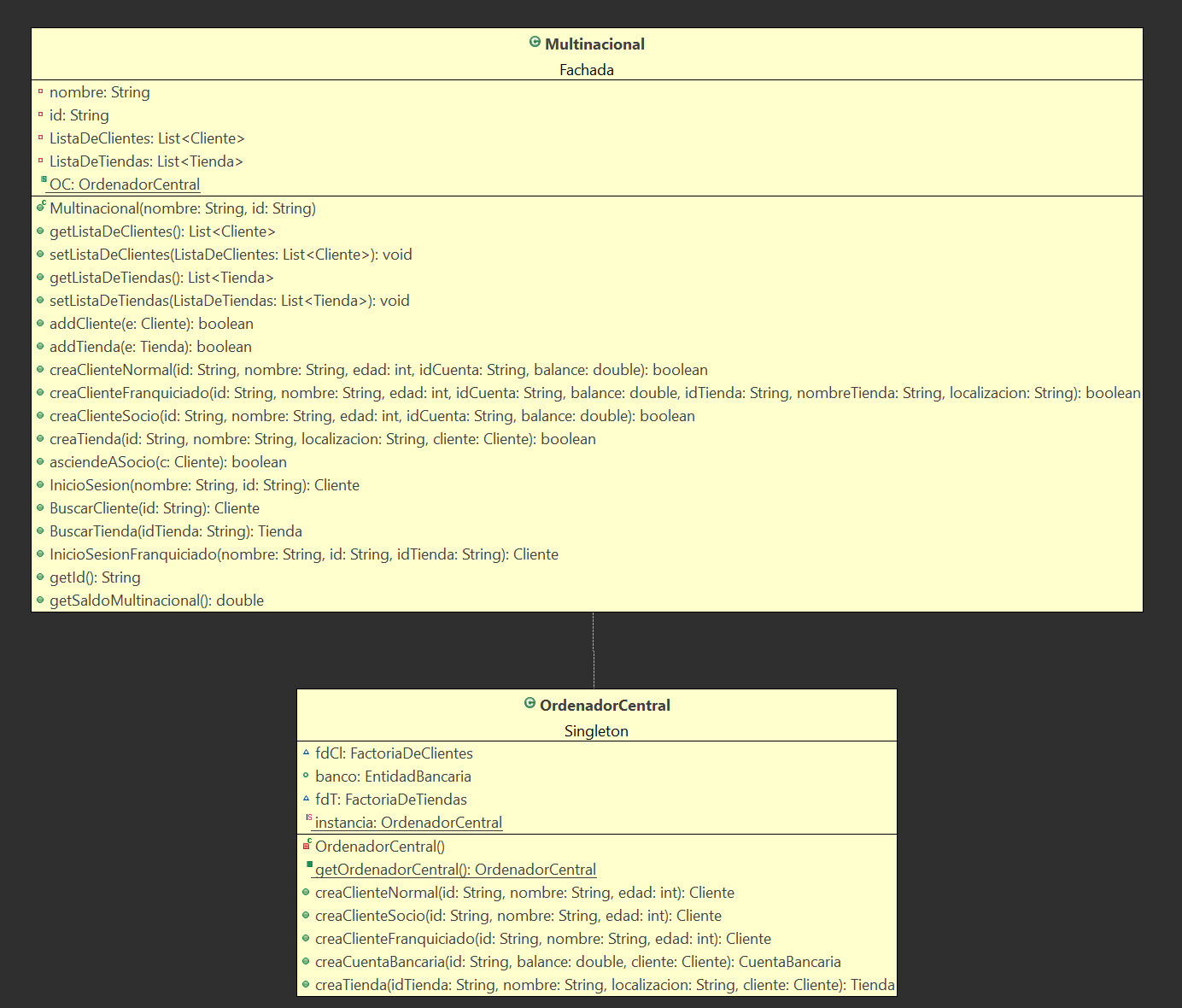
**Los grandes retos de diseño que este proyecto enfrenta pueden incluir:**

**Coordinación de datos:** Dado que hay múltiples clases y componentes involucrados en la gestión de clientes, tiendas y cuentas bancarias, asegurar una correcta coordinación y consistencia de los datos puede ser un desafío. El diseño debe garantizar que los datos se mantengan actualizados y coherentes en todo el sistema.

**Escalabilidad:** Si se espera que el sistema maneje un gran número de clientes, tiendas y transacciones, es importante considerar la escalabilidad del diseño. El proyecto debe ser capaz de manejar eficientemente grandes volúmenes de datos y transacciones sin degradar el rendimiento.

**Mantenibilidad y extensibilidad:** Dado que el proyecto puede requerir cambios y adiciones en el futuro, es fundamental diseñar el sistema de manera que sea fácil de mantener y extender. Esto implica seguir principios de diseño sólidos, como la modularidad, la cohesión y el bajo acoplamiento.

1. **Información del fragmento donde aparece el patrón**



**Relación entre las clases OrdenadorCentral y Multinacional**

La clase OrdenadorCentral, ubicada en el paquete Singleton, actúa como un componente central en un sistema o aplicación. Aquí está la función principal de esta clase y sus métodos:

**Singleton:** La clase OrdenadorCentral utiliza el patrón de diseño Singleton para garantizar que solo haya una instancia de la clase en todo el programa. El método getOrdenadorCentral() se utiliza para obtener la instancia única de OrdenadorCentral. Esto puede ser útil cuando se necesita una única instancia centralizada para coordinar y gestionar diferentes componentes del sistema.

**Creación de clientes:** La clase proporciona métodos para crear diferentes tipos de clientes, como clientes normales, clientes socios y clientes franquiciados. Estos métodos utilizan una instancia de FactoriaDeClientes para crear las instancias correspondientes de clientes y devuelven el objeto de cliente creado.

**Creación de cuentas bancarias:** La clase proporciona un método creaCuentaBancaria() para crear una cuenta bancaria. Este método utiliza una instancia de EntidadBancaria para crear la cuenta bancaria con un identificador, un saldo inicial y el cliente asociado. Devuelve el objeto de cuenta bancaria creado.

Creación de tiendas: La clase proporciona un método creaTienda() para crear una tienda. Este método utiliza una instancia de FactoriaDeTiendas para crear la tienda con un identificador, un nombre, una ubicación y el cliente asociado. Además, establece la tienda recién creada en la instancia del cliente franquiciado correspondiente. Devuelve el objeto de tienda creado.

En resumen, la clase OrdenadorCentral centraliza la creación de clientes, cuentas bancarias y tiendas, utilizando instancias de otras clases relacionadas. Además, garantiza que solo haya una instancia de OrdenadorCentral en todo el programa, lo que puede ser beneficioso para mantener un estado bien definido que gestiones las operaciones del sistema.

De esta forma, se podría resumir la estructura de la clase OrdenadorCentral de la siguiente manera:

* La clase maneja un identificador public.
* Posee 3 atributos declarados: fdCL, banco y fdT, cada una corresponde a las diferentes factorías mencionadas con anterioridad.
* Posee el constructor privado OrdenadorCentral() que impide la creación directa de instancias de esta clase fuera de la misma y el método estático getOrdenadorCentral() encargado de permitir el acceso a la instancia única de OrdenadorCentral.
* Diversos métodos para la creación de objetos relacionados con el funcionamiento de la multinacional (clientes, cuentas bancarias y tiendas).

Adicionalmente, para poder contextualizar mejor la clase OrdenadorCentral, se referirá a la clase Multinacional, ubicada en el paquete Fachada, en la que se hace uso de la estructura de OrdenadorCentral.

Inicialmente, se evidencia que se realiza una importación de la referencia estática de la clase OrdenadorCentral como “OC”, obtenida mediante el método getOrdenadorCentral(). Luego, en el constructor de Multinacional, se observa el instanciamiento del OrdenadorCentral, por medio del método anterior. En gran parte del código, se evidencia diversidad de métodos que permiten el acceso y modificación a listas de clientes y tiendas. Ahora, con relación al OrdenadorCentral, su relación se encuentra principalmente en los métodos de creación de objetos, pues en cada uno de estos se está haciendo uso de métodos pertenecientes a la instancia OC de la clase OrdenadorCentral. Por otra parte, existen métodos que dan la opción de ascender a un usuario a socio y métodos para buscar clientes y tiendas. Finalmente, retomando la conexión con el OrdenadorCentral, se observan métodos para obtener información, cómo lo es la obtención del ID y el saldo total de la multinacional, por medio de la instancia OC.

De esta forma, queda un poco más clara la importancia y el funcionamiento de la clase OrdenadorCentral, brindando funcionalidades principalmente respecto a creación y posterior obtención información de objetos.

1. **Información general sobre el patrón**

El patrón Singleton es uno de los patrones de diseño más conocidos y se utiliza para garantizar que una clase tenga una única instancia en todo el programa. Esto significa que no se puede crear más de una instancia de esa clase y proporciona un punto de acceso global a dicha instancia.

En el patrón Singleton, la clase en cuestión tiene un constructor privado para evitar que se cree directamente una instancia fuera de la clase. En su lugar, la clase proporciona un método estático para obtener la única instancia, generalmente llamado getInstance() o similar. Este método verifica si la instancia ya ha sido creada y, si es así, la devuelve. Si la instancia no ha sido creada, se crea una nueva y luego se devuelve.

El patrón Singleton se utiliza en situaciones en las que solo se requiere una única instancia de una clase para coordinar acciones en todo el programa. Algunos casos comunes de uso incluyen:

**Conexiones a bases de datos:** Cuando se necesita una única conexión a una base de datos en una aplicación, el patrón Singleton puede garantizar que solo haya una instancia de la clase de conexión en todo el programa.

**Registros de eventos:** En sistemas de registro o eventos, puede ser útil tener una única instancia que maneje la creación y escritura de registros. El patrón Singleton asegura que solo haya una instancia de la clase de registro.

1. **Información del patrón aplicado al proyecto**

Algunas de las principales características que brindan información sobre el patrón son:

**Clase Singleton:** La clase OrdenadorCentral está diseñada como un Singleton, lo que significa que solo puede haber una única instancia de esta clase en todo el programa. Esto se logra mediante la implementación de un constructor privado y una variable estática privada instancia, que almacena la única instancia de la clase.

**Método estático de obtención de la instancia:** La clase OrdenadorCentral proporciona un método estático getOrdenadorCentral() que permite acceder a la única instancia de la clase. Este método comprueba si la instancia ya existe; si es así, la devuelve, y si no, crea una nueva instancia y la devuelve.

**Uso del Singleton en otras clases:** Otras clases en el proyecto, como la clase Multinacional en el paquete Fachada, hacen uso de la clase OrdenadorCentral accediendo a su instancia a través del método getOrdenadorCentral(). Al tener una única instancia compartida, todas las clases que utilizan el OrdenadorCentral pueden acceder y utilizar los mismos datos y métodos.

**Beneficios del patrón Singleton:** El patrón Singleton garantiza que solo exista una instancia de la clase OrdenadorCentral en todo el programa. Esto puede ser beneficioso en situaciones en las que se requiere un único punto de acceso global para compartir información o recursos entre diferentes componentes del sistema. En este caso, el Singleton asegura que todos los clientes y tiendas de la multinacional bancaria accedan a la misma instancia de OrdenadorCentral, lo que facilita la coordinación y el acceso a los datos compartidos.

1. **¿Por qué tiene sentido haber utilizado el patrón en ese punto del proyecto? ¿Qué ventajas tiene?**

El uso del patrón Singleton en la clase OrdenadorCentral del proyecto tiene sentido por las siguientes razones y ventajas:

**Garantiza una única instancia:** El patrón Singleton asegura que solo haya una instancia de la clase OrdenadorCentral en todo el programa. Esto es útil en el contexto de una multinacional bancaria, ya que se requiere un componente centralizado y coordinado para gestionar y controlar las operaciones del sistema. Al tener una única instancia, se evitan problemas de inconsistencia y se asegura que todas las partes del sistema estén trabajando con la misma instancia de OrdenadorCentral.

**Acceso global:** La implementación del patrón Singleton proporciona un punto de acceso global a la instancia de OrdenadorCentral a través del método getOrdenadorCentral(). Esto permite que otras partes del sistema puedan obtener y utilizar la instancia de manera sencilla y sin tener que pasar explícitamente la instancia como parámetro o compartirla de otra manera. Facilita la interacción con el componente central y simplifica el diseño y la implementación del sistema.

**Estado centralizado:** Al tener una única instancia de OrdenadorCentral, se logra un estado centralizado en el sistema. Esto significa que todas las operaciones relacionadas con la creación de clientes, cuentas bancarias y tiendas se realizan a través de esta instancia central. Esto facilita la coordinación y gestión de estas operaciones, ya que se evita la dispersión de lógica y datos en diferentes instancias y componentes del sistema.

**Control de creación:** El patrón Singleton permite controlar la creación de la instancia de OrdenadorCentral. El constructor privado y el método estático getOrdenadorCentral() garantizan que la instancia se cree una sola vez y se devuelva la misma instancia en sucesivas llamadas. Esto asegura la consistencia y coherencia de la instancia en todo el sistema y evita la creación innecesaria de múltiples instancias.

En resumen, el uso del patrón Singleton en la clase “OrdenadorCentral” del proyecto de una multinacional bancaria tiene sentido porque garantiza una única instancia centralizada para coordinar y gestionar las operaciones del sistema. Proporciona acceso global a esta instancia, logra un estado centralizado y controla la creación de la instancia. Estas ventajas contribuyen a un diseño más coherente, eficiente y mantenible del sistema.

1. **¿Qué desventajas tiene haber utilizado el patrón en ese punto del proyecto?**

Aunque el patrón Singleton puede ser útil en muchos casos, también tiene algunas posibles desventajas cuando se utiliza en el contexto específico de la clase OrdenadorCentral en el proyecto de la multinacional bancaria. Algunas de las desventajas potenciales incluyen:

**Acoplamiento fuerte:** El uso del patrón Singleton puede llevar a un acoplamiento fuerte entre las clases que dependen de la instancia de OrdenadorCentral. Esto se debe a que todas las clases que necesitan acceder a OrdenadorCentral deben conocer su existencia y utilizar el método getOrdenadorCentral(). Esto puede dificultar la modificación o sustitución de OrdenadorCentral en el futuro, ya que requeriría cambios en múltiples partes del sistema.

**Dificultad para realizar pruebas unitarias:** El patrón Singleton puede dificultar las pruebas unitarias, ya que las dependencias de una clase que utiliza OrdenadorCentral estarán acopladas a la instancia única de OrdenadorCentral. Esto puede hacer que sea complicado simular diferentes escenarios y estados para las pruebas unitarias, ya que la instancia única de OrdenadorCentral se comparte entre las pruebas.

**Posible violación del principio de responsabilidad única:** La clase OrdenadorCentral puede tener múltiples responsabilidades, como la creación de clientes, cuentas bancarias y tiendas. Esto podría violar el principio de responsabilidad única, ya que la clase se está encargando de varias tareas distintas. Si el sistema se vuelve más complejo y las responsabilidades de OrdenadorCentral aumentan, podría ser difícil mantener y extender esta clase sin afectar otras partes del sistema.

**Dificultad para escalar horizontalmente:** El patrón Singleton es adecuado cuando se requiere una única instancia en todo el programa. Sin embargo, si en el futuro se desea escalar horizontalmente el sistema, es decir, ejecutar múltiples instancias en diferentes nodos o servidores, el uso del patrón Singleton puede dificultar esta escalabilidad, ya que cada instancia tendría su propia copia de OrdenadorCentral en lugar de compartir una única instancia centralizada.

1. **¿De qué otras formas se le ocurre que se podrían haber solucionado, en este caso particular, los problemas que resuelve el patrón?**

Algunas alternativas al patrón Singleton podrían ser:

**Inyección de dependencias:** En lugar de utilizar un Singleton para acceder a la instancia de OrdenadorCentral, se puede aplicar el principio de inyección de dependencias. En lugar de que las clases dependan directamente de OrdenadorCentral, se les puede proporcionar una instancia de OrdenadorCentral como parámetro o a través de un contenedor de dependencias. Esto permite una mayor flexibilidad y facilita las pruebas unitarias al poder proporcionar implementaciones alternativas de OrdenadorCentral o simularlo en las pruebas.

**Patrón Factory:** En lugar de tener una única instancia de OrdenadorCentral, se puede utilizar el patrón Factory para crear instancias de OrdenadorCentral según sea necesario. Esto permite crear múltiples instancias si es necesario y evita las limitaciones de un Singleton. Cada instancia de OrdenadorCentral tendría su propio estado y se pueden gestionar independientemente.

**Patrón Observer:** En lugar de depender de una única instancia de OrdenadorCentral, se pueden utilizar patrones de diseño como Observer para notificar a las clases interesadas sobre los cambios relevantes en el sistema. En este enfoque, las clases interesadas se suscriben a los eventos o notificaciones generados por diferentes componentes del sistema, lo que les permite realizar acciones en función de esos eventos sin depender directamente de OrdenadorCentral.

**Descomposición en componentes más pequeños:** En lugar de tener una única clase OrdenadorCentral que realiza múltiples tareas, se puede considerar una descomposición en componentes más pequeños y especializados. Por ejemplo, se podrían tener clases separadas para la gestión de clientes, cuentas bancarias y tiendas. Esto ayuda a cumplir con el principio de responsabilidad única y facilita la comprensión y el mantenimiento del código.