Trabajo Práctico Nº1	Unidad 1
Modalidad: Semi -Presencial	**Estratégica Didáctica:** Trabajo individual.
Metodología de Desarrollo: Det. docente	**Metodología de Corrección:** Via Classroom.
Carácter de Trabajo: Obligatorio – Con Nota	**Fecha Entrega:** A confirmar por el Docente.

ALUMNA:

DOMINGUEZ, SANDRA

POO y Patrones TP 004

PATRONES DE COMPORTAMIENTO

Marco Teórico:

Responder el siguiente cuestionario en función de la bibliografía Obligatoria.

1. Describir los Tipos de Patrones de Comportamiento

Los patrones de comportamiento tratan con algoritmos, se encargan de una comunicación efectiva y la asignación de responsabilidades entre objetos.

Chain of responsibility: permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena. Se evita acoplar al emisor y receptor de una petición dando la posibilidad a varios receptores de consumirlo. Cada receptor tiene la opción de consumir esa petición o pasárselo al siguiente dentro de la cadena.

Command: es un patrón de diseño de comportamiento que convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación te permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y soportar operaciones que no se pueden realizar. Son objetos que encapsulan una acción y los parámetros que necesitan para ejecutarse.

Iterator: es un patrón de diseño de comportamiento que te permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol,etc.). Se utiliza para poder movernos por los elementos de un conjunto de forma secuencial sin necesidad de exponer su implementación específica.

Mediator: te permite reducir las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador.
Objeto que encapsula cómo otro conjunto de objetos interactúan y se comunican entre sí.

Observer: Los objetos son capaces de suscribirse a una serie de eventos que otro objetivo va a emitir, y serán avisados cuando esto ocurra.

State: Permite modificar la forma en que un objeto se comporta en tiempo de ejecución, basándose en su estado interno.

Strategy: Permite la selección del algoritmo que ejecuta cierta acción en tiempo de ejecución.

Template Method: Especifica el esqueleto de un algoritmo, permitiendo a las subclases definir cómo implementan el comportamiento real.

Visitor: Permite separar el algoritmo de la estructura de datos que se utilizará para ejecutarlo. De esta forma se pueden añadir nuevas operaciones a estas estructuras sin necesidad de modificarlas.

2. ¿Qué tipos de Problema Resuelve?

Los patrones de diseño son **soluciones para problemas típicos y recurrentes** que nos podemos encontrar a la hora de desarrollar una aplicación. Si **la forma de solucionar ese problema se puede extraer, explicar y reutilizar** en múltiples ámbitos, entonces nos encontramos ante un patrón de diseño de software.

Aunque nuestra aplicación sea única, tendrá partes comunes con otras aplicaciones: acceso a datos, creación de objetos, operaciones entre sistemas etc. Podemos solucionar problemas utilizando algún patrón, ya que son soluciones probadas y documentadas por multitud de programadores. Los patrones ayudan a estandarizar el código, haciendo que el diseño sea más comprensible para otros programadores.

Los patrones de diseño nos ayudan a cumplir muchos de los principios o reglas de diseño. Programación SOLID, control de cohesión y acoplamiento o reutilización de código son algunos de los beneficios que podemos conseguir al utilizar patrones. Si queremos desarrollar aplicaciones robustas y fáciles de mantener, es recomendable cumplir ciertas reglas de diseño. Si bien su aplicación no es obligatoria, es muy necesario aplicar tanto las reglas como los patrones de diseño a consciencia, adecuando su uso a su estricta necesidad.

3. Dar ejemplos de Problema – Patrones de Comportamiento.

STRATEGY

En el sistema de tickets, la asignación de asistentes telefónicos se realiza mediante diferentes políticas:
Lineal: Se organiza a los empleados en una lista de manera tal que cada vez que entra un nuevo pedido de ayuda se le asigna al siguiente en la lista.
Equitativa: El sistema asigna el nuevo pedido de ayuda al empleado que tiene la menor cantidad de chats asignados.
Temática: Cada pedido de ayuda se relaciona con un tema (por ejemplo: no me puedo registrar en el sistema; no puedo cambiar la contraseña; mi cuenta está bloqueada; etc.). Cada empleado está capacitado para atender pedidos relacionados con ciertos temas. El sistema asigna los pedidos según el tema del que se traten.
el sistema de atención al público debe lograr que:

- Se puedan agregar, modificar y eliminar políticas de asignación de chats.
- Los administradores del sistema puedan configurar el sistema con la política que mejor se ajusta a sus necesidades, si necesidad de detener y reiniciar el sistema.

COMMAND

Una máquina en un peaje que recibe monedas y billetes. Establecer la comunicación entre la máquina en la estación de peaje y el módulo que implementa la lógica de pago en efectivo, el cual ocultaba la comunicación con el hardware de la máquina.
Crear una barra de herramientas con unos cuantos botones para varias operaciones de un editor de texto. En la cual hay que implementar menús contextuales, atajos y otros elementos, esto duplicará el código de la operación en muchas clases, o peor aún, hará menús dependientes de los botones.

ITERATOR

La mayoría de las colecciones almacena sus elementos en simples listas, pero algunas de ellas se basan en pilas, árboles, grafos y otras estructuras complejas de datos.
Independientemente de cómo se estructure una colección, debe aportar una forma de acceder a sus elementos de modo que otro código pueda utilizar dichos elementos.
Debe haber una forma de recorrer cada elemento de la colección sin acceder a los mismos elementos una y otra vez.
MEDIATOR

4. ¿Qué rol juega la Interfaz en los Patrones de Comportamiento? ¿Qué facilita?

Marco Práctico:

Desarrollar el Problema expresado debajo e Implementarlo en C++.

REPASO STRATEGY - ¿Qué es?

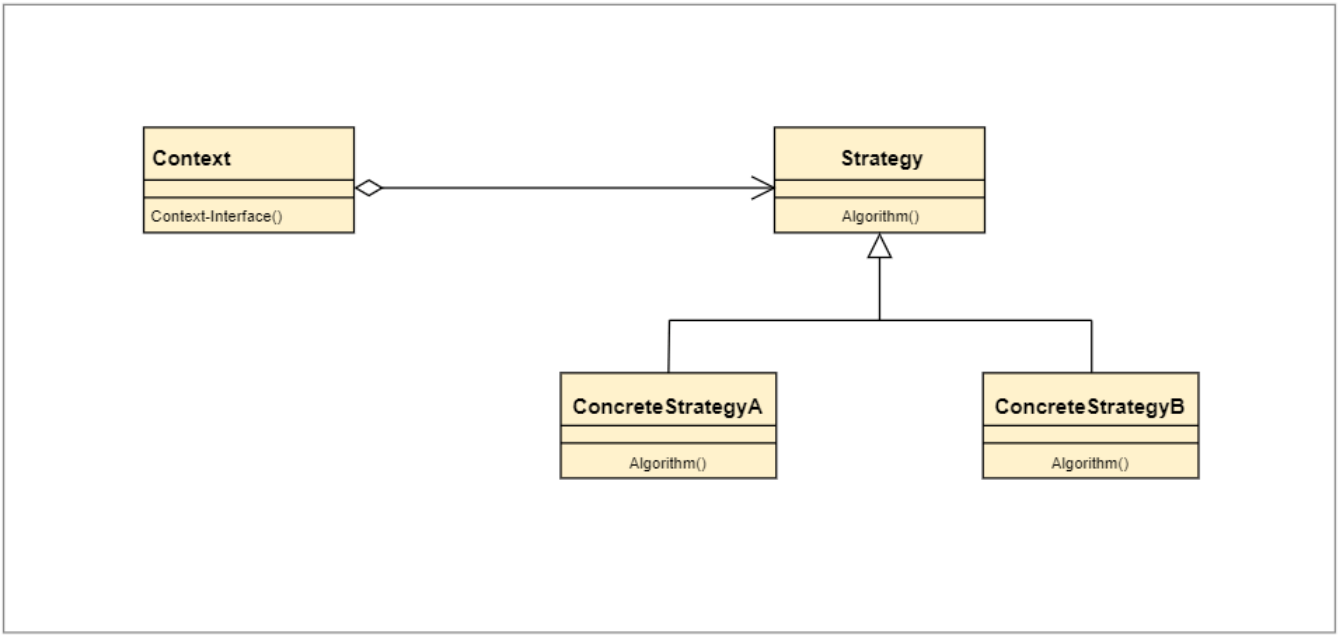
¿En qué consiste el strategy pattern?

El *strategy pattern* pertenece a los llamados ***behavioural patterns* o patrones de comportamiento**, que equipan un software con diferentes métodos de resolución. Estas **estrategias** consisten en una familia de algoritmos que están separados del programa real y son autónomos, es decir intercambiables. Un *strategy pattern* también incluye algunas pautas y ayudas para los desarrolladores. Por ejemplo, los *strategy patterns* describen cómo construir u organizar un grupo de clases y crear objetos. Una característica especial del patrón *strategy* es que un **comportamiento variable de programas y objetos** también se puede realizar en el tiempo de ejecución de un software.

¿Cómo es la representación UML de un strategy pattern?

Los *strategy patterns* se diseñan generalmente con el lenguaje de modelado gráfico UML (*Unified Modeling Language).* Sirven para visualizar los patrones de diseño con una **notación estandarizada** y utilizan caracteres y símbolos especiales. El UML establece distintos tipos de diagramas para la programación orientada a objetos. Para representar un *strategy pattern*, se suelen utilizar los llamados diagramas de clase con al menos **tres componentes básicos:**

- Context (contexto o clase de contexto)
- Strategy (estrategia o clase de estrategia)
- ConcreteStrategy (estrategia concreta)



Estructura básica de un patrón strategy en UML con tres componentes básicos: Context (clase principal), Strategy (interfaz) y ConcreteStrategies (algoritmos externalizados y especificaciones de solución para la resolución de problemas concretos).

Con el *strategy pattern*, los componentes básicos asumen funciones especiales. Los patrones de comportamiento de la **clase Context** se almacenan en diferentes clases Strategy. Estas clases separadas contienen los algoritmos llamados ConcreteStrategies. Utilizando una **referencia** interna, el contexto puede acceder a las variables de computación externalizadas (ConcreteStrategyA, ConcreteStrategyB, etc.) si lo necesita. No interactúa directamente con los algoritmos, sino con una interfaz.

La **interfaz Strategy encapsula** las variantes de cálculo y puede ser implementada simultáneamente por todos los algoritmos. Para la interacción con Context, la interfaz genérica proporciona solo un método para activar los algoritmos de ConcreteStrategy. La interacción con Context incluye, además del acceso a Strategy, **intercambio de datos**. La interfaz Strategy también participa en los cambios de estrategia, que pueden tener lugar, además, en el tiempo de ejecución del programa.

Repaso

La encapsulación impide el acceso directo a los algoritmos y a estructuras de datos internos. Una instancia externa (Client, Context) solo puede acceder a los cálculos y funciones a través de interfaces definidas y, además, solo puede acceder a los métodos y elementos de datos de un objeto que sean relevantes para ella.

¿Cuáles son las ventajas y e inconvenientes del strategy pattern?

Las ventajas de usar un *strategy pattern* son más evidentes desde la **perspectiva de un programador y administrador de sistemas.** El desglose en módulos y clases autónomas ayuda a **estructurar mejor el**

código del programa. Puesto que los módulos estén delimitados en nuestra aplicación de ejemplo, la tarea del programador será más sencilla. Así pues, se puede limitar el alcance de la clase Navigator mediante la externalización de las estrategias y se puede prescindir de la creación de subclases en Context.

Las dependencias internas de los segmentos se mantienen dentro de los límites de un código más reducido y claramente definido. Por ello, **los cambios tienen menos efecto**, es decir, no suelen conllevar más cambios (que requieren mucho tiempo) en la programación. En algunos casos, los cambios derivados incluso pueden descartarse por completo. Los segmentos de código más claros **también pueden mantenerse mejor a largo plazo** y el diagnóstico y la resolución de problemas se facilitan.

Asimismo, **el manejo se hace más sencillo**, ya que la aplicación de ejemplo se puede equipar con una interfaz fácil de usar. Los usuarios pueden usar los botones para **controlar** el comportamiento del programa (cálculo de la ruta) de forma variable y elegir entre las opciones de forma sencilla.

Puesto que el Context de la aplicación de navegación solo interactúa con una interfaz que encapsula los algoritmos, es **independiente de la aplicación concreta de los algoritmos** individuales. Por lo tanto, si se modifican los algoritmos o se introducen nuevas estrategias posteriormente, no es necesario **cambiar el código de Context**. Esto permite ampliar las funciones de cálculo de ruta con estrategias concretas adicionales para rutas en avión, transporte marítimo y tráfico de larga distancias. Las nuevas estrategias solo tienen que implementar la interfaz de Strategy correctamente.

Los *strategy patterns* simplifican la difícil programación del software orientado a objetos gracias a otra de sus virtudes: permiten el **diseño de software reutilizable (módulos) que se puede implementar repetidamente** y cuyo desarrollo se considera particularmente difícil. Esto significa que las clases Context relacionadas también podrían utilizar las estrategias externalizadas para calcular las rutas a través de la interfaz y ya **no tendrían que aplicarlas por sí mismas**.

A pesar de sus muchas ventajas, el *strategy pattern* también tiene algunos **inconvenientes**. Debido a su estructura más compleja, el diseño del software puede crear redundancias e ineficiencias en la comunicación interna. Por ejemplo, la interfaz *strategy* genérica, que todos los algoritmos deben aplicar por igual, a veces puede **acabar sobredimensionada**.

Un ejemplo: una vez que Context ha creado e inicializado ciertos parámetros, los pasa a la interfaz genérica y al método definido en esta. Sin embargo, la estrategia que se aplique en última instancia no necesita necesariamente todos los parámetros comunicados de Context y, por lo tanto, no los procesa. Así, **la interfaz proporcionada no siempre se utiliza de manera óptima en el *strategy pattern*** y a veces se producen transferencias de datos innecesarias, con el consiguiente esfuerzo de comunicación.

En la aplicación, también existe una estrecha dependencia **interna entre el cliente y las estrategias**. Client hace la selección y solicita la estrategia concreta mediante el comando de activación (en nuestro ejemplo, el cálculo de la ruta a pie), por lo que debe conocer las ConcreteStrategies. Por lo tanto, el patrón de diseño *strategy* **solo debe utilizarse si los cambios de estrategia y comportamiento** son importantes para el uso y la funcionalidad de un software.

Naturalmente, los inconvenientes mencionados se pueden compensar parcialmente o minimizar. Por ejemplo, el número de instancias de objetos que pueden producirse en grandes cantidades en el *strategy pattern* puede reducirse si se **aplican a un *flyweight pattern*.** La medida también tiene un efecto positivo en los requisitos de eficiencia y en la memoria de la aplicación.

¿Dónde se utiliza el patrón strategy?

Como patrón de diseño básico en el desarrollo de software, el *strategy pattern***no está limitado a un solo ámbito de aplicación.** Lo más importante a la hora de escoger este patrón de diseño es la **naturaleza del**

problema que se quiera resolver. El patrón *strategy* es ideal para software que ha de resolver tareas y problemas variables, opciones de comportamiento y cambios.

Por ejemplo, los programas que ofrecen **diferentes formatos de almacenamiento de archivos** o varias funciones de clasificación y búsqueda utilizan el *strategy pattern*. En el ámbito de la **compresión de datos** también se utilizan programas que emplean diversos algoritmos de compresión basados en este patrón de diseño. Esto permite, por ejemplo, convertir de forma versátil los vídeos a un formato de archivo que ahorre espacio o restaurar archivos comprimidos (por ejemplo, archivos ZIP o RAR) a su estado original mediante estrategias especiales de descompresión. Otro ejemplo es el **almacenamiento de un documento** o archivo gráfico en diferentes formatos.

El patrón de diseño *strategy* también se utiliza en el desarrollo e **implementación de software de juegos**, que, por ejemplo, tienen que reaccionar con flexibilidad a las situaciones cambiantes del juego durante el tiempo de ejecución. Los diferentes personajes, los equipos especiales, los patrones de comportamiento de los personajes o sus movimientos especiales pueden almacenarse en forma de ConcreteStrategies.

Otra área de aplicación de los *strategy patterns* es el **software de control**.Mediante el intercambio de ConcreteStrategies, los segmentos de cálculo pueden adaptarse fácilmente a grupos ocupacionales, países y regiones. También los programas que traducen **los datos a diferentes formatos gráficos** (por ejemplo, como gráficos de líneas, circulares o de barras) utilizan *strategy patterns*.

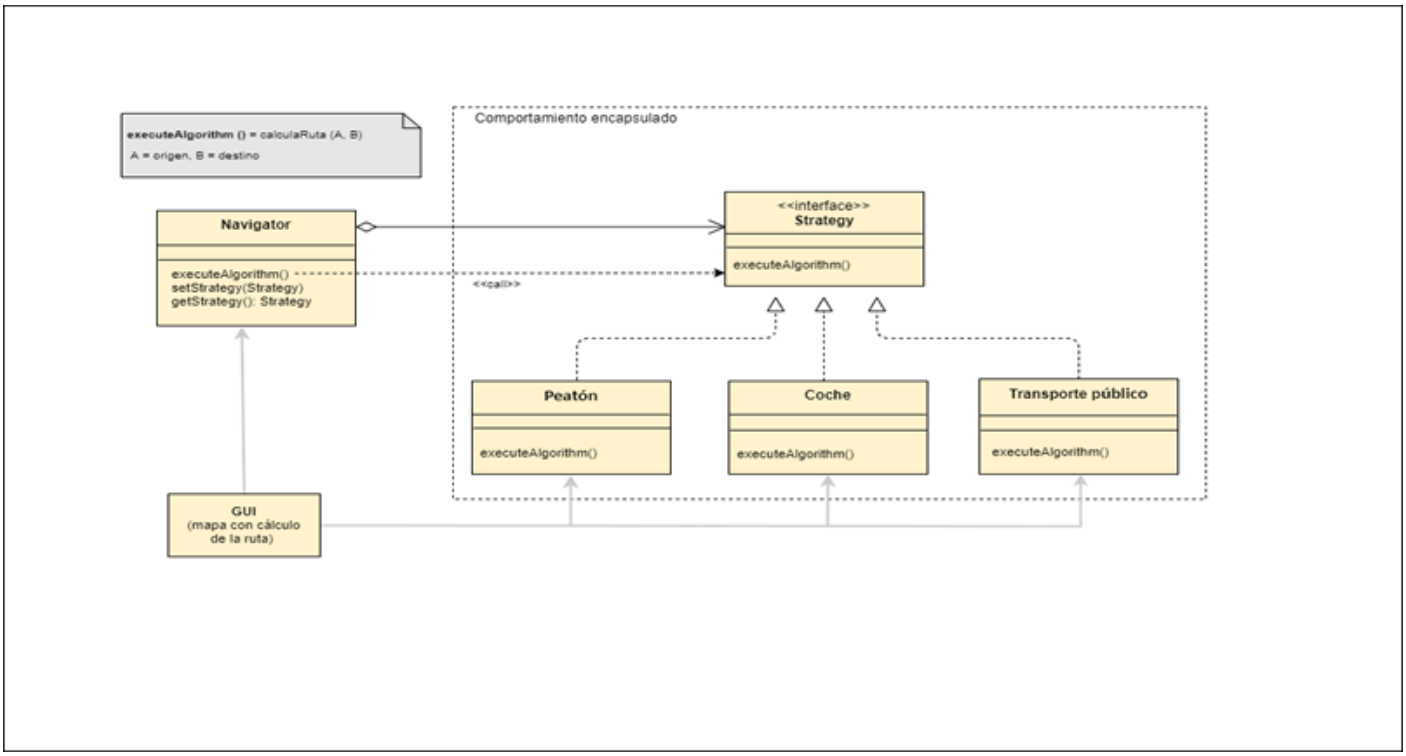
Se pueden encontrar aplicaciones más específicas de los *strategy patterns* en la **biblioteca estándar de Java** (Java API) y **en los conjuntos de herramientas de la interfaz gráfica de Java** (por ejemplo, AWT, Swing y SWT), que utilizan un gestor de diseño en el desarrollo y la generación de interfaces gráficas de usuario. Este gestor puede aplicar diferentes estrategias para la disposición de los componentes durante el desarrollo de la interfaz. El *strategy pattern* se utiliza también en **sistemas de base de datos**, controladores de dispositivos y programas de servidores.

Modulo Practico

Desarrollar una Aplicación que deba **calcular una ruta** basada en los medios de transporte habituales. El usuario puede elegir entre tres opciones:

- Peatón (ConcreteStrategyA)
- Coche (ConcreteStrategyB)
- Transporte público (ConcreteStrategyC)

Si se transfieren estas especificaciones a un gráfico UML, se puede observar la **estructura y la función** del *strategy pattern*:



Strategy pattern en el diseño de una aplicación de navegación en UML: la clase Context recibe un comando del Client (cliente) de la aplicación (Consola) y luego establece la estrategia requerida. La interacción con los algoritmos de estrategia encapsulados (ConcreteStrategies) tiene lugar a través de la interfaz Strategy. Para poder solicitar una estrategia concreta, el Client debe conocer todas sus implementaciones.

En nuestro ejemplo, el **cliente o Client** es la Interfaz de Consola (Podría ser la GUI) de una aplicación de navegación con opciones para el cálculo de la ruta. Si el usuario hace una selección (o pulsa un botón en una GUI), se calcula una ruta concreta. El **Context (clase Navigator)** tiene la tarea de calcular y mostrar una serie de puntos de control en el mapa. La clase Navigator tiene un método para cambiar la estrategia de enrutamiento activa. Los botones permiten cambiar fácilmente entre los medios de transporte.

Si, por ejemplo, se activa un comando correspondiente con la Opcion **peatonal del Client**, se solicita el servicio "Calcular la ruta peatonal" (ConcreteStrategyA). El método *executeAlgorithm()* (en nuestro ejemplo, el método: *calculaRuta(A, B))* acepta un origen y un destino y devuelve un conjunto de los puntos de control de la ruta. Context acepta el comando del Client y decide la estrategia apropiada **basándose en las directivas previamente definidas (*policy*)** (*setStrategy: peatonal).* Mediante *Call*, delega la solicitud al objeto Strategy y a su interfaz.

Con *getStrategy()*, la estrategia actualmente seleccionada se almacena en Context (clase Navigator).
Los **resultados de los cálculos de ConcreteStrategy se utilizan para el procesamiento posterior** y en la visualización por consola de la ruta en la aplicación de navegación. Si el usuario elige una ruta diferente, por ejemplo, haciendo clic en la Opcion "Coche", Context cambia a la estrategia solicitada (ConcreteStrategyB) e inicia un nuevo cálculo a través de otra *llamada.* Al final del procedimiento, se devuelve **una descripción de ruta modificada para el coche.**

Se solicita Implementar este Ejem plo en C++.

La Cátedra.
Lic. Oemig José Luis.