Trabajo Final - 2do. Sem. 2025

INTEGRANTES:
- Julian Derudi - julianderudi9@gmail.com
- Maximiliano Jerochim - maxjerochim98@gmail.com
- Santiago Staffora - s.staffora@gmail.com

Para registrar las órdenes de importación o exportación creamos un objeto Turno que almacene la información relevante (como el chofer y el identificador del camión) en sus variables de instancia. Turno será usado como TurnoConsignee por quienes quieran importar y cómo TurnoShipper por quienes quieran exportar.
Esto nos permite trabajar polimórficamente con los métodos *verificarDemora* y *realizarOperacion* cuya implementación varía según si se trata del turno de un shipper o un consignee.

Para *verificarDemora* trabajamos asumiendo:
1. Un camión puede llegar a un turno tan temprano como desee.
2. Sin importar la hora de llegada de un camión, siempre se le compara con la hora de su turno más temprano.

Nuestra terminal también cuenta con un objeto ListaTurnos que busca modelar la lista de las órdenes pedidas por distintos clientes. Esta clase implementa una lista de Turno y está pensada para ser la única responsable de operar sobre la lista de órdenes, delegando así la responsabilidad de la terminal.

Para realizar la operación correspondiente según si se trata de un camión enviado por un shipper o un consignee realizamos un doble dispatch a través de ListaTurnos donde enviamos el camión y la propia terminal, ListaTurnos delega la tarea al turno más temprano de ese camión, y finalmente el turno le indicará a la terminal si se trata de un retiro o depósito de carga y será esta quien la realice.
Esto implementa efectivamente un patrón visitor donde la terminal es la visitante del turno.

Los containers pueden recibir distintos servicios mientras se encuentran en la terminal, esto es modelado como una variable de instancia en los containers de tipo *Lista<Servicio>*.
Los containers de tipo Reefer son los únicos que admiten servicio de electricidad, así que cuando cualquier otro tipo de container desea agregar un servicio primero verifica que este no sea un servicio de electricidad.

Para calcular el coste de los servicios de un Container el mismo recorre su lista de servicios y suma el coste de cada uno individualmente a través de *calcularCoste.* Calcular el coste de los servicios se trabaja como un patrón strategy donde cada subclase de servicio es, en definitiva, una estrategia distinta de cómo se calcula el costo del servicio.

En la implementación del buque utilice el patrón state debido a que el buque puede pasar por diferentes fases las cuales representan estados que hacen que el comportamiento del buque cambie de una forma u otra según el estado en que se encuentre. La aplicación del patrón consistió en delegarle al estado, osea a la fase del buque, el comportamiento. Esto ayudó a que el código no solo sea más entendible sino también a poder hacer restricciones de que operaciones puede o no puede hacer el buque según en qué estado se encuentre. Por ejemplo, el método informar() que informa a los shippers de sus exportaciones está solamente permitido en la fase Outbound. Esto se logra porque todas las fases del buque implementan una misma interfaz que contiene sus propios métodos, pero cada una responde a su manera. Algunas con excepciones si el método no está permitido, y otras con su propia implementación.

Por otro lado el buque conoce a la terminal portuaria que debe llegar como destino, la línea naviera a la que pertenece, los clientes que tiene asociados con sus containers, y su distancia de la terminal.

La terminal contiene **lineas navieras**, las cuales le proveen a esta de **circuito maritimos** para realizar viajes hacia otras terminales.

Por otro lado la terminal hace uso de un **Motor de Busqueda** para poder filtrar los circuitos que cumplen con cierta/s condición/es. Esto está implementado haciendo uso del patrón strategy y además del **composite**, Cumpliendo los siguiente roles, MotorDeBusqueda es el Context del strategy, y Filtro es el Strategy, siendo las clases que lo implementan las StrategyConcretes. Y para el patron Composite aplicado Filtro es el **Component** y las clases que lo implementan son las **Leat**, salvo FiltroCombinacion que es el **Composite**.

En este Sistema hay otro **Strategy**, el del MejorCircuito, cumpliendo este con el rol de **Strategy** y las clases que lo implementan sus **StrategyConcrete**. Esta clase devuelve el mejorCircuito dependiendo de cada una que lo implemente de cierta forma, está abierto a la extensión, pero cerrado a la modificacion, cumpliendo los principios SOLID.