Trabajo Final – Programación Con Objetos II - 2do. Sem. 2023

Terminal Portuaria

Programación con Objetos II – Comisión 1

- Bertolini Agaras, Victor Hugo
- Lattenero, Ignacio Francisco
- Medina, Ivan Angel

Emails:

- victor_bertolini_agaras@yahoo.com.ar
- ignaciolattenero@gmail.com
- ivanangelmedina@gmail.com

Decisiones de Diseño.

Para adherirnos al principio de Responsabilidad Única, hemos optado por la creación de la clase Deposito. Esta clase se encarga de las tareas relacionadas con el manejo de los contenedores y la verificación de los camiones, funciones que originalmente pertenecían a la clase Terminal Gestionada. De esta manera, la clase Terminal Gestionada puede centrarse exclusivamente en los procesos de importación y exportación, así como en la búsqueda de rutas, que son parte integral del proceso de exportación. Al introducir la clase Deposito, hemos logrado delegar responsabilidades, permitiendo que la Terminal Gestionada se ocupe únicamente de sus tareas principales.
Además, hemos introducido otras clases como Factura y Mail, que cumplen un propósito similar al de la clase Deposito. Estas clases se encargan de tareas que, aunque en un esquema más amplio podrían considerarse responsabilidades de la Terminal Gestionada, hemos decidido delegar para mantener la coherencia con la idea central de esta clase. La Terminal Gestionada se concibe como una entidad dedicada a la importación y exportación de carga a través de buques. Al delegar tareas secundarias a las clases Factura, Mail y Deposito, logramos que la Terminal Gestionada se mantenga fiel a su propósito principal.

Detalles de Implementación.

Como parte de los detalles de implementación, hemos decidido que al instanciar un objeto Consignee, se le asigna automáticamente una OrdenImportacion. Esto se basa en la suposición de que cuando se organiza el proceso de exportación desde una TerminalNormal, ya se ha creado una OrdenImportacion que está en posesión del Consignee. Además, hemos hecho la suposición de que el buque siempre está al tanto de la Terminal Gestionada. Esta suposición es necesaria para realizar ciertos cálculos en los estados del Patrón de Diseño State, que gestiona los estados del Buque (Inbound, Working, etc.). Por último, hemos asumido que, en los tramos de un circuito, todos están conectados de manera secuencial. Es decir, el destino final del primer tramo coincide con el origen del tramo siguiente, y así sucesivamente.

Patrones de Diseño.

En el marco de nuestro proyecto, hemos aplicado tres patrones de diseño específicos. El patrón State se ha utilizado para gestionar los diferentes estados del Buque. El patrón Strategy se ha implementado en la administración de los servicios y en la selección del mejor circuito. Por último, el patrón Template Method se ha empleado en la clase Factura.

En el contexto del patrón de diseño State, la clase Fase cumple el rol de State, que define la interfaz común para todos los estados concretos. La clase Buque actúa como el Context, manteniendo una referencia a uno de los estados concretos y permitiendo la transición entre ellos. Las clases Arrived,

Departing, Inbound, Outbound y Working son los ConcreteStates, cada una implementando un comportamiento asociado a un estado del Buque. Esta estructura permite cambiar el comportamiento del Buque de manera dinámica en función de su estado interno.

Del lado del patrón de diseño Template Method, la clase Factura actúa como la AbstractClass, proporcionando una estructura básica para los métodos de la factura. Dentro de esta clase, los métodos Desglose, imprimirFecha, imprimirServicios, imprimirCostoTramos y imprimirCostoTotal conforman los componentes del patrón.

El método Desglose es el Template Method, que define el esqueleto del algoritmo y llama a los otros métodos en un orden específico. Los métodos imprimirFecha y imprimirServicios son los Primitive Operations, que proporcionan implementaciones para las operaciones necesarias del algoritmo. Los métodos imprimirCostoTramos e imprimirCostoTotal son los Hook Operations, que proporcionan puntos de extensión donde las subclases pueden variar el algoritmo.

Las clases FacturaShipper y FacturaConsignee son las ConcreteClasses en este patrón. Estas clases implementan o sobrescriben los métodos definidos en la Factura (AbstractClass) para proporcionar las variantes del algoritmo. Este patrón permite cambiar partes del algoritmo sin cambiar la estructura general.

En el primero de los patrones Strategy usados la clase Servicio actúa como el Strategy, definiendo una interfaz común para todos los ConcreteStrategies. Los distintos servicios (Lavado, AlmacenamientoExcedente, Pesado, Electricidad) son los ConcreteStrategies, cada uno implementando un comportamiento específico.

La interfaz Orden es el Context en este patrón. Mantiene una referencia a un objeto Strategy y delega parte de su comportamiento al Strategy que tiene. Al cambiar el Strategy, Orden puede cambiar su comportamiento en tiempo de ejecución.
Finalmente, en el contexto del otro patrón Strategy, la clase abstracta Plan actúa como el Strategy, proporcionando una interfaz común para todos los ConcreteStrategies. El método operacion() en Plan es el comportamiento que las subclases PlanMenorTiempo, PlanMenorPrecioTotal y PlanMenorCantidadDeTerminales pueden personalizar según sus necesidades.
Estos patrones de diseño han sido fundamentales para estructurar nuestro código de manera eficiente y mantener la coherencia en nuestra implementación.