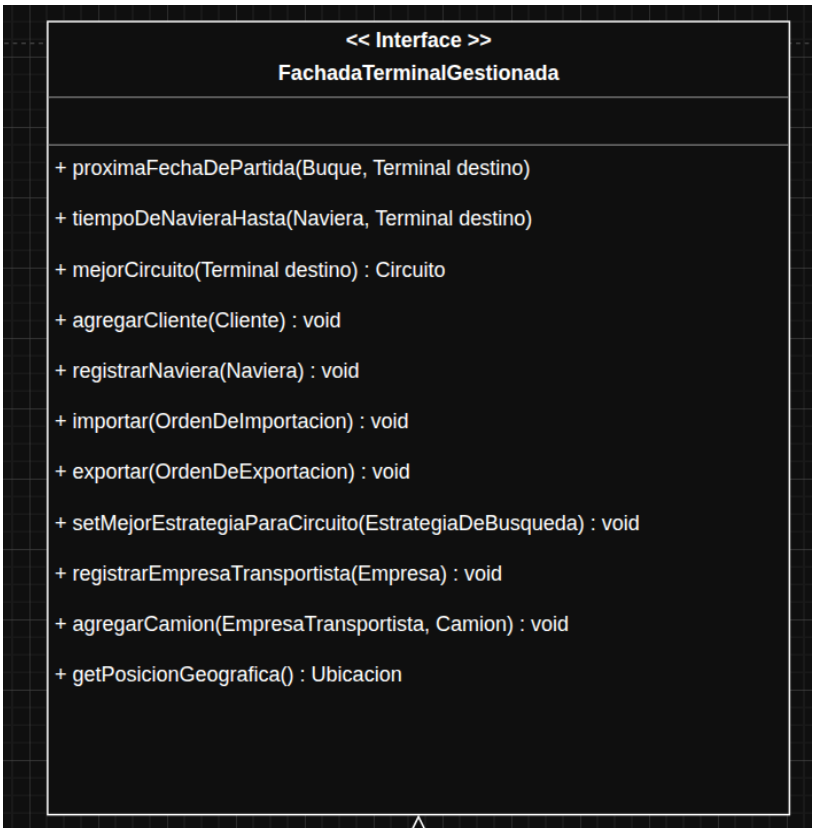
Documentación de software de una Terminal Portuaria

Capítulo I : La fachada

La terminal portuaria implementa un patrón de diseño llamado Facade que consiste en ofrecer al usuario una interface de funcionalidades.



Si usted decide ser un usuario de este sistema, podrá hacer uso de esta interfaz sin preocuparse por quien se encarga de realizar la tarea, simplemente envía el mensaje y tendrá la respuesta que describe como resultado.

Es una de las ventajas de este patrón, ocultar la complejidad de los subsistemas que tiene como colaboradores.

Por otro lado, si está desarrollando nuevas funcionalidades para este software, en los capítulos siguientes tendrá más detalles respecto a implementaciones.

Capítulo II : La terminal gestionada implementa la fachada

Participantes:
Fachada(TerminalGestionada)
Clases del subsistema: (Logística, Warehouse, GestionTerrestre)
Implementan las funcionalidades del subsistema, realizan las operaciones encargadas por la Fachada.

Responsabilidad única

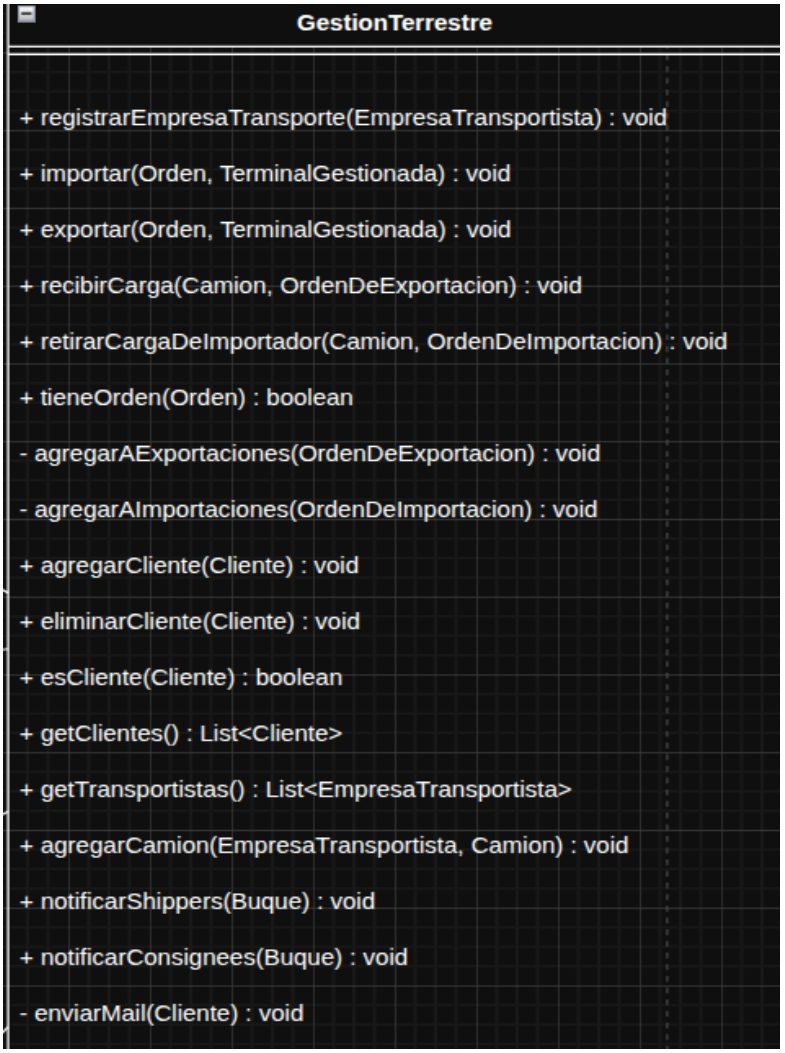
La terminal gestionada es quien implementa la fachada y tendrá como única responsabilidad saber quien de sus colaboradores es el encargado de realizar dicha tarea.

Posee tres grandes colaboradores:

- Gestión Terrestre: Encargado de clientes, camiones, empresas de transporte y ordenes de comercio exterior.
- Logística : Encargada de manejar buques, navieras, sus circuitos y viajes.
- Warehouse: Encargada del manejo de cargas y servicios.

Capítulo III: La gestión terrestre

Revisemos sus responsabilidades:
Encargado de clientes, camiones, empresas de transporte y ordenes de comercio exterior. Parece mucha responsabilidad para una sola clase, pero también sabe cómo delegar tareas a colaboradores. Podríamos decir que “es un Facade dentro de un facade”.
(Véase Inception/ El origen, de no entender esta referencia)



Hay algunas operaciones que son muy triviales, como lo son:

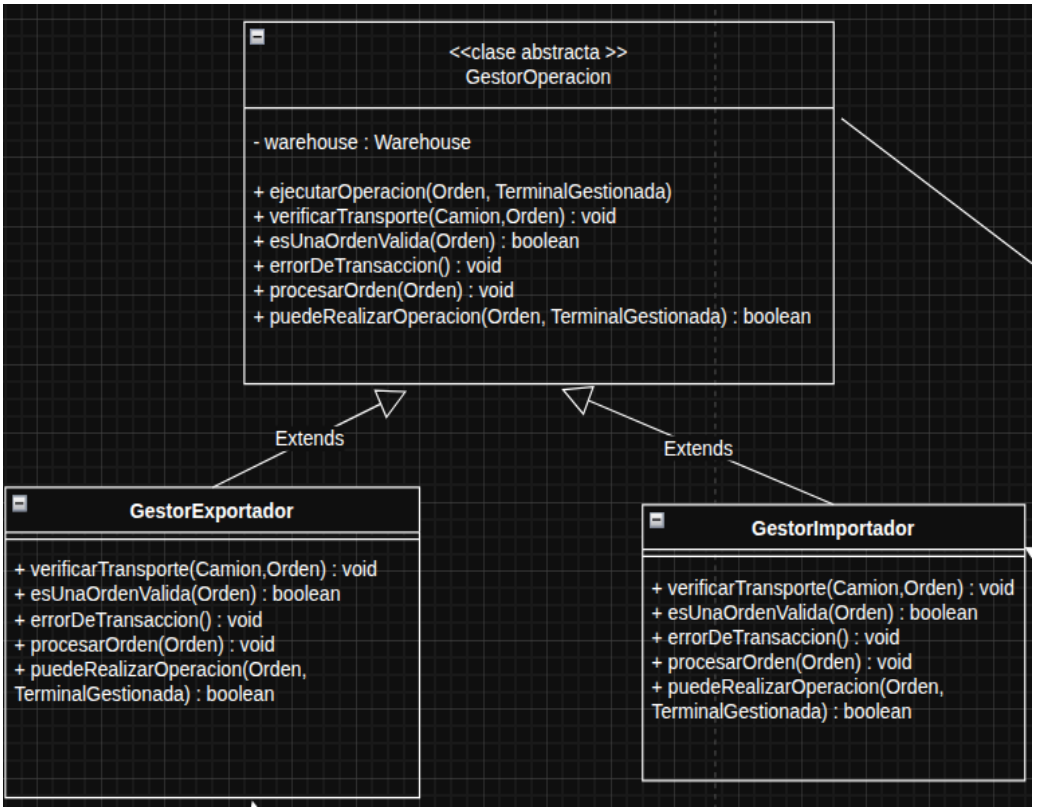
- agregarCliente(Cliente), getCliente(Cliente) ,eliminarCliente(Cliente)..son operaciones basicas sobre una lista de dicho elemento.
- como también lo es registrarEmpresaDeTransporte()
- Como también verificar que un elemento exista en una colección, como lo es tieneOrden(Orden)

Los métodos exportar e importar son más interesantes, y para estas funcionalidades, la clase GestionTerrestre sabe quienes deben realizar estas tareas.Tiene dos colaboradores para cada funcionalidad, para importar se encargará el GestorDeImportacion, y para exportar el GestorDeExportacion. Más detalles sobre estos gestores a continuación.

Capítulo III: Patrón template para ejecutar operaciones de importación y exportación.

Tendremos una clase GestorDeOperacion para estructurar el comportamiento de los gestores.

El patrón template definirá el esqueleto del algoritmo en una clase llamada GestorDeOperacion, quien definirá los pasos y las clases GestorDeImportacion/ GestorDeExportacion sobrescriben pasos sin cambiar la estructura.



Participantes:
Clase abstracta(GestorOperacion)
Clase concreta: GestorExportador, GestorImportador.

Método plantilla: ejecutarOperacion(Orden, TerminalGestionada)

Colaboraciones:
Las clases concretas se basan en la clase abstracta para implementar los pasos del algoritmo que esta en el método plantilla.

Documentación de Diseño: TP Terminal Portuaria

Metodología de Desarrollo:

La implementación se guio principalmente por la metodología TDD (Test-Driven Development). Esta decisión se tomó para asegurar el cumplimiento del requisito de alta cobertura de tests (95%) y para construir un diseño robusto, modular y mantenible.

Cada módulo (Buque, Carga, Servicios) se desarrolló comenzando por los tests (ciclo Rojo-Verde-Refactor), lo que forzó la creación de interfaces/clases claras y la aplicación de principios como la Inversión de Dependencias (DIP). Tambien se aprovechó bastante el uso de Mocks (Mockito) en los tests.

Patrones de Diseño Utilizados:

Se identificaron y aplicaron varios patrones de diseño "GoF" para resolver los problemas planteados.

<u>Patrón State (Estado):</u>

Este es el patrón central para la gestión del ciclo de vida del buque.

Problema: Un Buque pasa por una secuencia de fases (Outbound, Inbound, Arrived, Working, Departing) , y su comportamiento cambia drásticamente en cada fase (por ej., no puede iniciarTrabajo si está en Outbound).

Decisión de Diseño: Se implementó el patrón State para encapsular el comportamiento de cada fase en su propia clase.

Roles:

Contexto (Context): La clase Buque. Mantiene una referencia a su estado actual (ej: private EstadoBuque fase;). Delega las operaciones (como actualizarPosicion o iniciarTrabajo) a este objeto de estado.

Estado (State): La interfaz EstadoBuque. Define la interfaz común para todas las fases (ej: actualizarPosicion(Buque buque), iniciarTrabajo(Buque buque), depart(Buque buque)).

Estados Concretos (ConcreteState): Las clases Outbound, Inbound, Arrived, Working y Departing. Cada una implementa la lógica específica de esa fase.

Detalle de Implementación: Las transiciones de estado son manejadas por los propios ConcreteState. Por ejemplo, Outbound implementa la lógica de actualizarPosicion y, si detecta que distancia < 50km, es responsable de ejecutar buque.setFase(new Inbound()).

Las notificaciones a la terminal (ej: terminal.notificarArriboInminente(buque)) también se disparan desde el estado concreto en el momento de la transición .

Patrón Composite (Composición):

Este patrón se utilizó para modelar el contenido de los containers (Bill of Lading).

Problema: El TP especifica que un B/L puede ser simple (un producto) o un "BL especial que agrupa a otros BLs" . El sistema debe poder tratar a ambos de manera uniforme (ej: para calcularPesoTotal()).

Decisión de Diseño: Se implementó el patrón Composite para crear una jerarquía de árbol.

Roles:

Componente (Component): La interfaz BillOfLading. Define las operaciones comunes (ej: getPesoTotal(), getTiposDeProducto()).

Hoja (Leaf): La clase BLSimple. Representa un B/L indivisible. Implementa getPesoTotal() devolviendo su peso propio.

Compuesto (Composite): La clase BLCompuesto. Mantiene una List<BillOfLading>. Implementa getPesoTotal() sumando recursivamente el resultado de sus hijos (blsAgrupados.stream().mapToDouble(...).sum()).

Detalle de Implementación: Se integró este patrón con la jerarquía de Carga. La clase Carga tiene un atributo BillOfLading. Para cumplir las reglas de negocio, los constructores de Reefer y Tanque validan que el B/L recibido no sea una instancia de BLCompuesto (lanzando una IllegalArgumentException si lo es).

Decisiones de Diseño y Principios SOLID:

Además de los patrones, se tomaron decisiones clave para asegurar un diseño cohesivo y desacoplado.

Principio de Responsabilidad Única (SRP):

Servicios: Cada clase de servicio (ej: ServicioElectricidad) encapsula la lógica compleja de su propio cálculo de costo.

Detalle de Implementación (Cálculo de Costos):

ServicioElectricidad: Utiliza java.time.ChronoUnit.HOURS.between() para calcular las horas de conexión y multiplicarlas por el reefer.getConsumoKwHora() y el precio por KW .

ServicioAlmacenamiento: Compara la fechaRetiroEfectivo con la fechaLlegadaNotificada.plusHours(24). Si se excede, calcula los días de multa usando una división entera sobre las horas de demora.

Patrón Strategy (Estrategia)

El patrón Strategy se utilizó para resolver el problema de seleccionar el mejor circuito marítimo entre varias alternativas, dependiendo de distintos criterios de negocio (menor tiempo, menor precio, menor cantidad de tramos)

Problema:

La clase Logística debía decidir cuál circuito es el más conveniente para llegar a una terminal destino. El criterio de selección no es fijo: puede variar según la necesidad del usuario o la política de la naviera.

Decisión de diseño:

Se implementó el patrón Strategy para encapsular cada criterio de selección en su propia clase. De esta forma, Logística delega la decisión a la estrategia activa, sin necesidad de conocer la lógica interna.

Contexto (Context): La clase Logistica. Mantiene una referencia a la estrategia actual (private EstrategiaDeBusqueda estrategia;) y delega la selección del circuito a este objeto.

Estrategia (Strategy): La interfaz EstrategiaDeBusqueda. Define el contrato común

Estrategias Concretas (ConcreteStrategy):

EstrategiaMenorTiempo: selecciona el circuito con menor duración total.

EstrategiaPrecioMasBajo: selecciona el circuito con menor costo.

EstrategiaCircuitoCorto: selecciona el circuito con menos tramos.

Cada clase implementa el método según su criterio específico.

Implementación Patron Visitor

Problema a Resolver

El sistema requería la generación de múltiples reportes (Muelle, Aduana, Buque) sobre la misma estructura de objetos (un Buque y sus Cargas). Cada reporte necesitaba:

* Información diferente (uno solo cuenta, otro lista IDs, otro pide fechas).

* Formatos de salida distintos (Texto Plano, HTML, XML).

* Implementar métodos como generarReporteMuelle() o generarReporteHTML() directamente en las clases Buque y Carga violaría el Principio de Responsabilidad Única (SRP), que ensuciaban un poco el dominio con lógica de formato.

Decisión de Diseño: Patrón Visitor

Se optó por el patrón Visitor porque permite agregar nuevos reportes en el futuro sin modificar las clases de dominio (Buque o Carga).

La implementación se realizó siguiendo TDD, lo que nos llevó a definir dos interfaces principales que forman el núcleo del patrón.

Roles (según Gamma et. al.)

Identificamos y asignamos los roles del patrón Visitor de la siguiente manera:

Visitor: IVisitorReporte

Define la interfaz para los reportes. Declara un método visit por cada tipo de elemento concreto que puede visitar (visitBuque(Buque buque), visitCarga(Carga carga)). También incluye un método getReporte() para extraer el resultado.

ConcreteVisitor: ReporteMuelle, ReporteAduana, ReporteBuque

Cada clase implementa IVisitorReporte.

Cada reporte acumula su resultado (usando un StringBuilder) mientras visita los elementos.

Element (Interfaz): IElementoVisitable

Define la operación accept(IVisitorReporte visitor) que permite al visitante operar sobre el elemento.

ConcreteElement: Buque y Carga

Ambas clases implementan IElementoVisitable.

Contienen los datos necesarios para los reportes (ej: nombre, ID, fechaArribo, listas de cargas).

ObjectStructure: La clase Buque

El Buque actúa como la estructura principal. Su implementación de accept() no solo se visita a sí mismo, sino que también es responsable de la recursión, iterando sobre sus listas (containersCargados y containersDescargados) y llamando a accept() en cada Carga hija.

Detalles de Implementación (TDD)

El desarrollo TDD de este patrón siguió varios pasos clave:

Hacer "Visitables" las Clases: Primero, usamos TDD para forzar a Buque y Carga a implementar IElementoVisitable y el método accept.

Implementación "Pull": Decidimos que los visitantes usarían un enfoque de "Pull" (extracción). El visitante, dentro de visitBuque, es quien activamente le pide al buque todos los datos (buque.getNombre(), buque.getContainersDescargados()) y se encarga de formatearlos.

Lógica del accept:

En Buque:

```
@Override
public void accept(IVisitorReporte visitor) {
   // 1. Visita al Buque (a sí mismo)
   visitor.visitBuque(this);

   // 2. Visita a los hijos (recorre ambas listas)
   for (Carga carga : this.containersCargados) {
      carga.accept(visitor);
   }
   for (Carga carga : this.containersDescargados) {
      carga.accept(visitor);
   }
}
```

En Carga:

```
@Override
public void accept(IVisitorReporte visitor) {
   // Se visita a sí mismo
   visitor.visitCarga(this);
}
```

Detalle del ReporteBuque (XML): Este reporte utilizó ambos métodos. visitBuque construyó la estructura <import> y <export>, y visitCarga (llamado desde el accept de Carga) construyó la línea <item>...</item> .