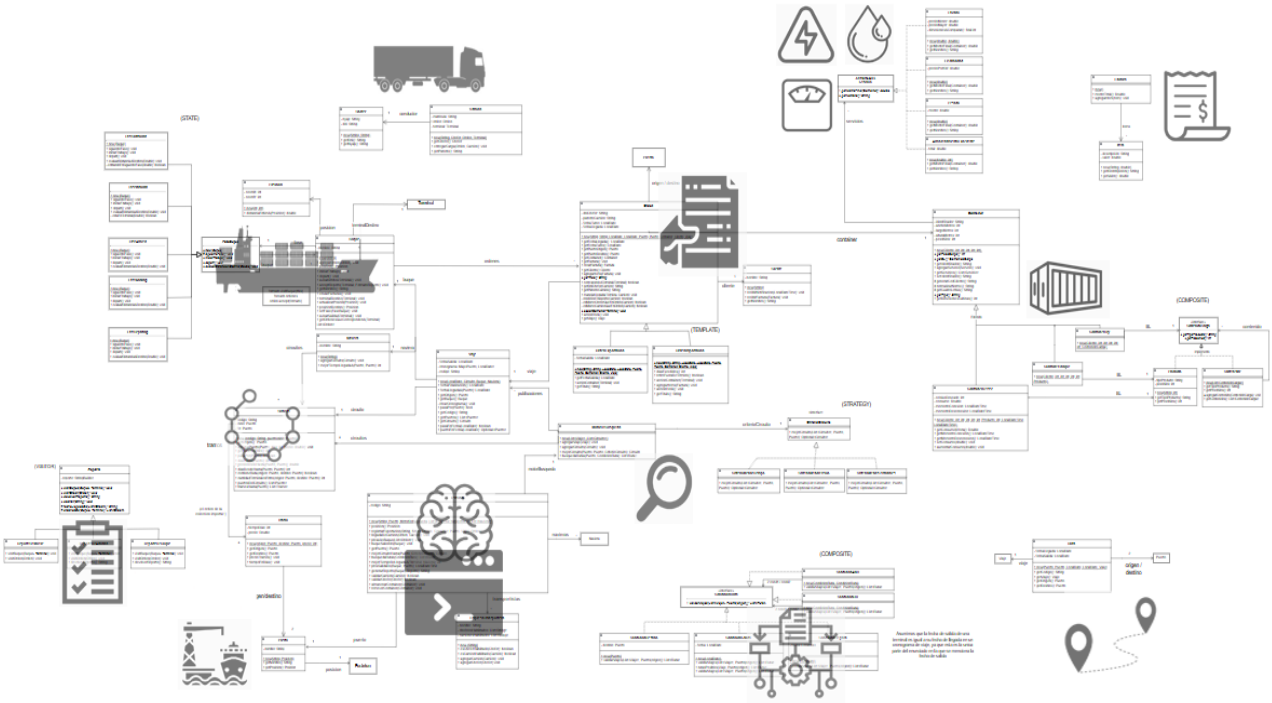
Informe de Trabajo Final

Terminal Portuaria - POO2



INTEGRANTES

Bogarín Leandro Javier - bogarleandro@gmail.com

Fuentes Joaquín – joaquinfuentes170@gmail.com

Quintana Ezequiel - e.n.de.quintana@gmail.com

Introducción

Nuestro Objetivo consiste en el diseño e implementación (Diagrama UML y código Java) de un sistema en el contexto de una terminal portuaria, dentro de un ecosistema portuario básico. Este ecosistema contiene elementos como líneas navieras, circuitos, conteiners, empresas transportistas, buques viajes, clientes (shippers y consignees), logística de exportación e importación, facturación de servicios, entre otros.

Todo el trabajo se centra en la gestión de una terminal, de la cual nacen varios procesos de:

- Exportación e importación
- Validación de elementos de transporte
- Búsqueda de viajes y circuitos en base a distintos criterios
- Almacenamiento de conteiners
- Generación de reportes
- Facturación de los servicios contratados.

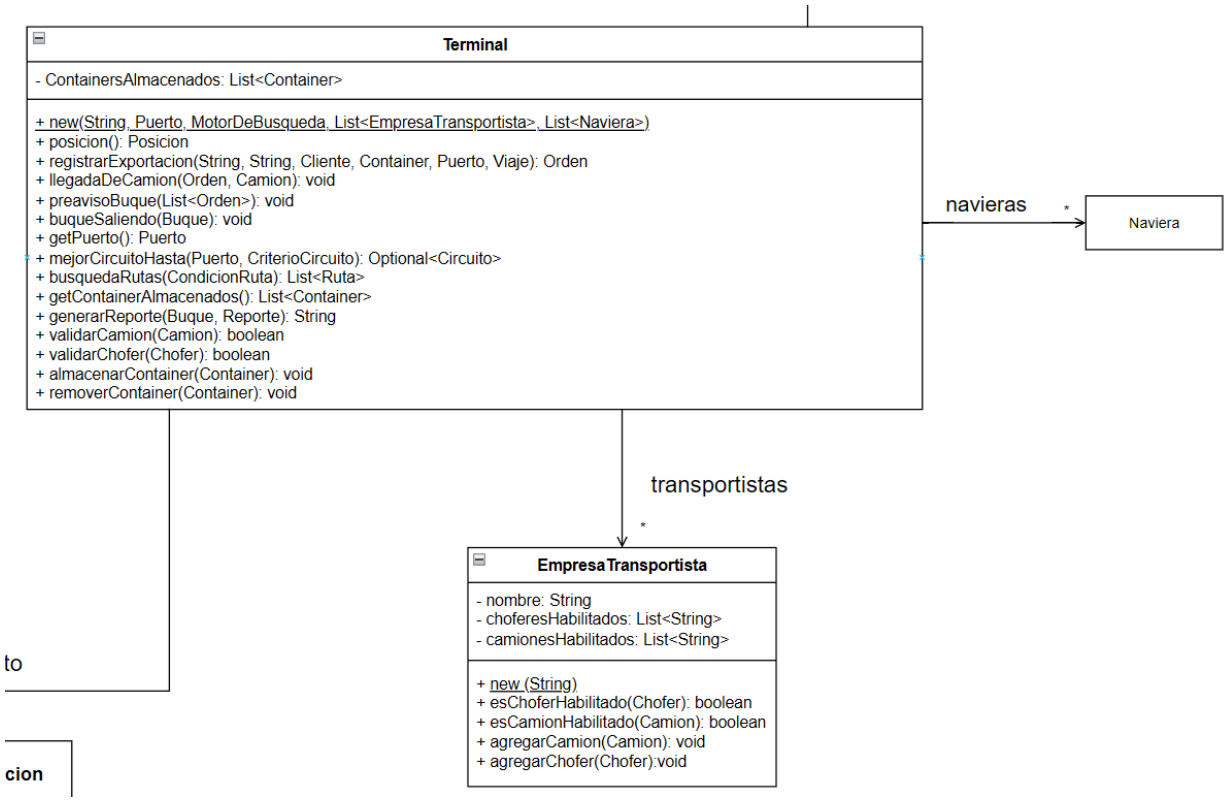
Además de la terminal, existen varias clases implementadas que aportan al flujo de funcionamiento de todos estos procesos.

Para la realización de este sistema aplicamos distintos patrones de diseño, teniendo en mente las distintas restricciones y consideraciones planteadas por el enunciado, como así también el uso de los principios SOLID a criterio.

Funcionalidades Implementadas de la terminal

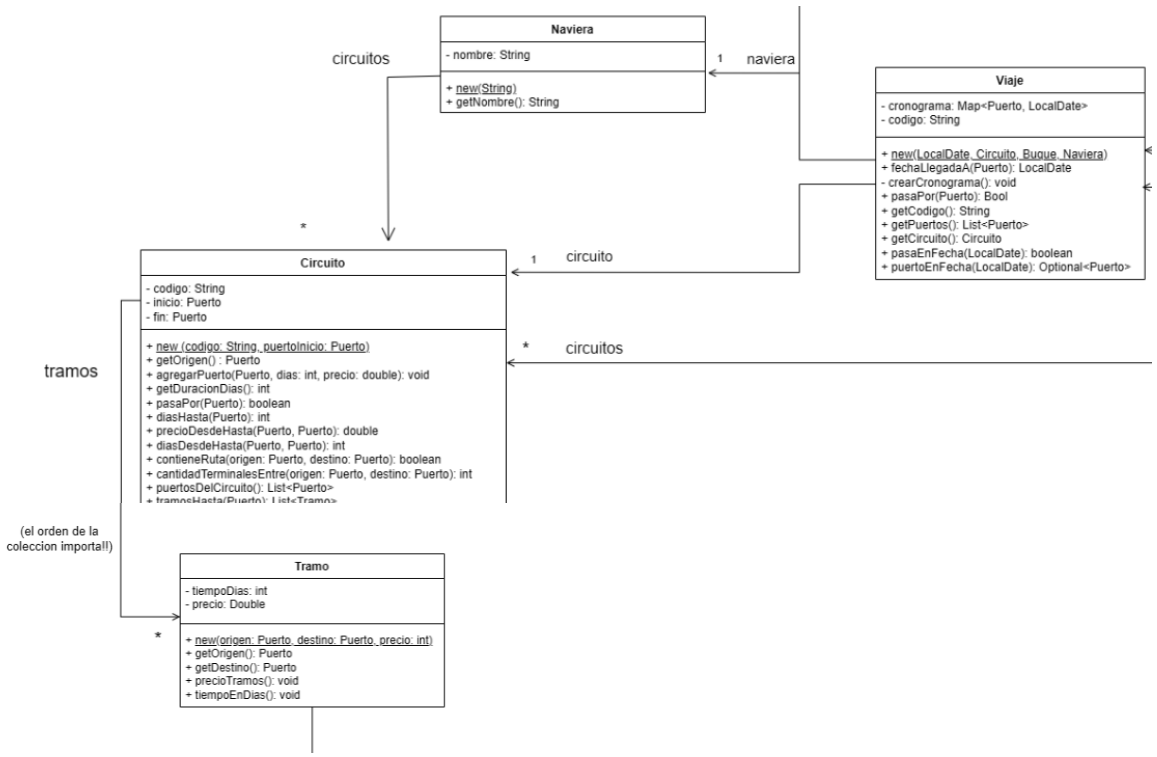
1. Registrar líneas navieras, shippers, consignees, empresas transportistas, camiones, conductores.

En nuestro diseño del trabajo la terminal registra y gestiona estos elementos atraves de distintas operaciones o relaciones directas. Esto es logrado registrando un conjunto de líneas navieras y de empresas transportistas. Además, la terminal interactúa con los clientes (shippers y consignees), a través de las órdenes de importación/exportación de cargas de clientes en viajes, realizados por un buque específico, pertinentes a la terminal. Los conteiners que transportan estas cargas pasan por un proceso de validación de los camiones que los llevan o retiran, para ser cargados o descargados en la misma.



2. Registrar los circuitos marítimos que incluyen a la terminal, considerando las fechas de las distintas programaciones de viajes (fechas de arribo a la propia terminal y a todas las demás de los circuitos que la incluyen).

Representamos a los circuitos como un conjunto de tramos, estos tramos representan un trayecto unidireccional de un puerto a otro, los cuales tienen una duración y un precio, ambos fijos. La programación de realizar un circuito en fechas determinadas es un Viaje, el cual genera su respectivo cronograma de fechas de arribo a cada puerto del circuito.



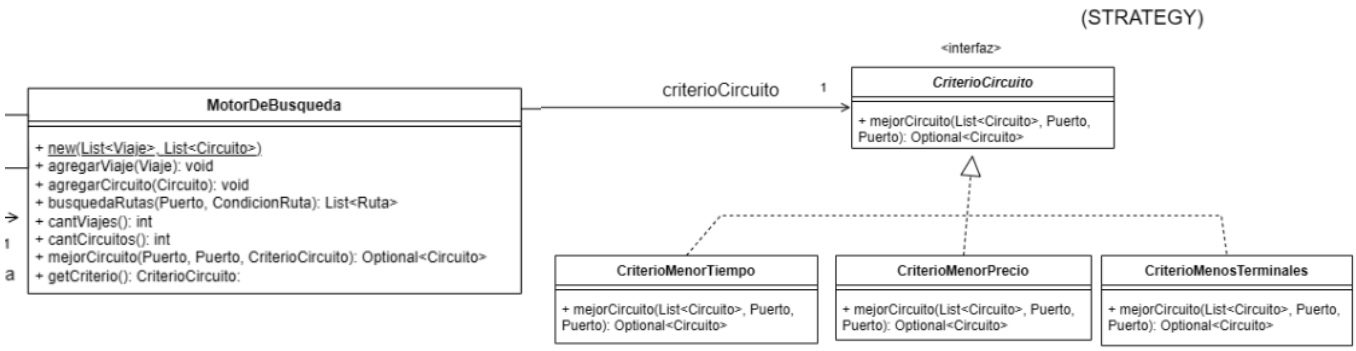
La terminal gestionada da a conocer los viajes y circuitos que le corresponden a través de su motor de búsqueda, es decir, los que pasan por su puerto. Dicho motor será mencionado más adelante.

3. Devolver el mejor circuito (o conjunto de tramos) que une a la terminal con un determinado puerto destino. El concepto de “mejor” debe poder ser seteado y cambiado dinámicamente en la terminal gestionada. Deben ofrecerse los siguientes tres, pero el modelo debe quedar abierto para poder agregar otros nuevos de manera flexible:
    a. Menor tiempo total de recorrido entre origen y destino.
    b. Menor precio total de recorrido entre origen y destino.
    c. Menor cantidad de terminales intermedias entre origen y destino.

La terminal puede devolver el mejor circuito a través del mensaje “mejorCircuitoHasta()”, el cual recibe un puerto de destino y un criterio para elegir el circuito. Se considera al origen para llegar a dicho destino como el puerto de la terminal gestionada.

Implementamos esta funcionalidad de la terminal con el patrón de diseño Strategy, considerando a los distintos “criterios” de circuito como las estrategias a las que se le delega el comportamiento.

La terminal posee esta funcionalidad, pero el motor de búsqueda es el encargado de registrar dicho criterio y de delegarle la tarea, dándole los circuitos que correspondan al trayecto especificado (desde la terminal gestionada hasta el destino dado).



Roles del patrón:

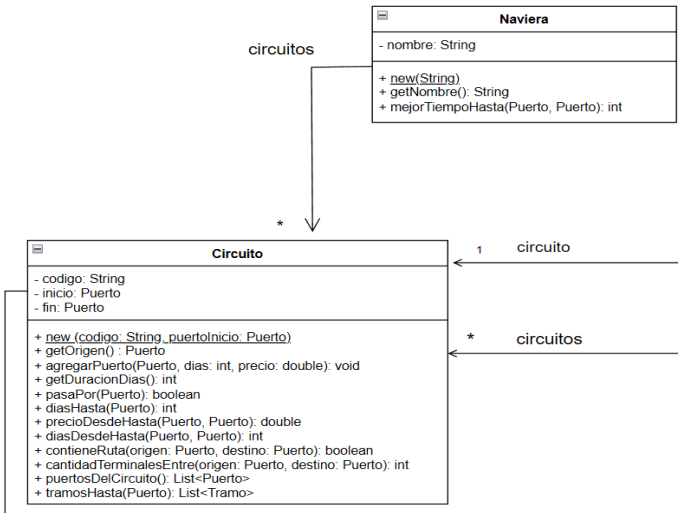
Contexto (MotorDeBusqueda): cumple el rol de “contexto”, siendo el que recibe por parámetro los distintos criterios y los utiliza para devolver el mejor circuito

Estrategia (CriterioCircuito): Interfaz de criterio de circuito, solo tiene que definir el método de devolver mejor circuito

EstrategiasConcretas (MenorTiempo-Precio-Terminales): Clases concretas de CriterioCircuito, cada una define el comportamiento del método implementado

4. Devolver cuánto tarda una naviera en llegar desde la terminal gestionada hacia otra terminal, independientemente de las fechas de los viajes programados.

La terminal conoce el mensaje “diasDeNavieraHasta()” el cual recibe una naviera y un destino, y devuelve el menor tiempo que ofrece una naviera para llegar desde la terminal gestionada hasta la terminal destino dada. En caso de que la naviera no tenga un circuito que realice dicho trayecto, devuelve 0 (cero).
Cada naviera conoce la colección de circuitos que le pertenecen y envía el mensaje “diasDesdeHasta()” a dichos circuitos para obtener cuantos días tardan en realizar un trayecto de un puerto a otro.



5. Devolver la próxima fecha de partida de un buque desde la terminal gestionada hasta otra terminal de destino.

La terminal gestionada implementa el método “salidaDeBuqueHasta()” que recibe como parámetro un buque y un puerto de destino, y devuelve la próxima fecha de salida de ese buque desde la terminal gestionada, y que a su vez llegue al puerto destino.

La terminal delega esta tarea al motor de búsqueda, considerando para esta función y otras, a la fecha de “salida” como la fecha de llegada a cierto puerto en el mismo viaje. Es decir, consideramos que un buque llega y se va de un puerto en el mismo día.

6. Ejecución de circuitos completos de exportación e importación de containers.

Exportación

Una terminal ofrece distintas opciones para ver viajes o rutas en base a distintos criterios, y además ofrece la funcionalidad de registrar exportaciones. Este mensaje recibe datos para la exportación, valida datos de origen y de destino, y crea la orden de exportación correspondiente para un trayecto de un container, en un viaje, por parte de un cliente. Al crear la orden se registra además, un turno para que el cliente deje su carga en la terminal. Al llegar el camión con dicha carga se verifica que cumpla las condiciones necesarias para pasar.

Importación

Cuando se aproxima el buque a la terminal, se les notifica a los clientes correspondientes de la llegada de su carga, y se les asigna un horario para retirarla. La asignación de dicho horario y el turno asignado para dejar la carga en la exportación no es contemplada en el trabajo.

Una vez avisado el cliente informa sobre el camión y chofer responsables de retirar la carga, y en caso de demorarse más de un día para retirarla en relación al turno asignado, se le acreditara el servicio por día de almacenamiento excedente.

El buque del viaje que realice dichos trayectos se encargara de manipular las órdenes y dar los avisos correspondientes según su distancia a la terminal. Esto lo hace atraves de distintas “fases” que implementamos usando el patrón de diseño State.

En cuanto a la facturación de servicios, estas facturas serán realizadas y enviadas a cada cliente correspondiente una vez que el buque se aleje de la terminal. Esto aplica tanto para clientes que importaban como para los que exportaban. Ambos clientes serán responsables de pagar los servicios aplicados al conteiner, y en el caso de los “Consignee” también deberán acreditar el valor del trayecto realizado en el viaje.

Logramos esto delegando el comportamiento de la validación de llegada de camión y la creación de factura a las clases concretas de Orden de Exportación e Importación.

7. Para el ejercicio de Reportes **, investigue el patrón visitor para su umplementación.

Para esta parte del trabajo implementamos el patrón de diseño Visitor como sugerido por el enunciado. La terminal gestionada implementa un método que permite indicar, por parámetro, el buque del cual se busca generar el reporte, y el tipo de formato de reporte que se busca generar. A través de la terminal, dicho buque será llamado para aceptar la visita del reporte, el cual recompilara toda la información necesaria a través de visitas al buque y las ordenes, y armara un String que represente dicho reporte.

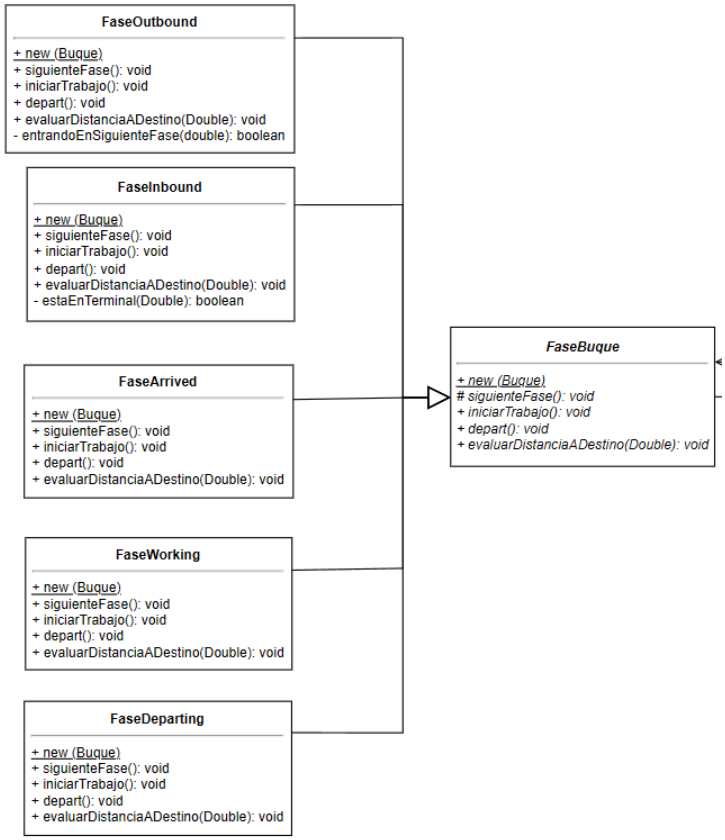
Nuestro diseño considera que un reporte se genera en relación a un buque en una terminal específica, registrando los contenedores que se cargan y descargan en la misma. Esta distinción la logramos gracias a las órdenes de exportación/importación. Cada clase concreta de orden sabe cuál es el criterio para verificar si corresponde a una terminal (puerto). Las ordenes de exportación saben que corresponden al puerto si dicho puerto figura en su origen, mientras que las ordenes de importación saben que corresponden al puerto si dicho puerto figura como su puerto destino. Las órdenes de exportación representan conteiners cargados mientras que las de importación representan conteiners que se descargan.

Todo lo anteriormente mencionado fueron aclaraciones sobre decisiones de diseño para las funcionalidades de la terminal que se enumeran al final del enunciado. El trabajo obviamente contiene otra gran cantidad de implementaciones para otras funcionalidades de todo el sistema, como por ejemplo la asignación de servicios, la lógica de filtrado de rutas o circuitos para las búsquedas, las fases del buque, los diferentes tipos de conteiners, entre otros.

A continuación mencionamos un resumen de los patrones utilizados en nuestro trabajo, con una breve descripción del contexto de cada uno, y las clases que lo conforman con sus respectivos roles en la implementación del patrón.

Patrón State en fases de buque:

En el apartado de “Fases de buque” del enunciado se habla sobre como el buque manejara distinto comportamiento dependiendo de la fase en la que se encuentre. Además se define un flujo de fases y los eventos que generan cambios en las mismas. Este es un claro ejemplo de una implementación con el patrón State.



Un buque siempre se encuentra en una fase y delega la mayoría de su comportamiento a dicha fase. Estas fases fluyen principalmente en base a la distancia del buque con su siguiente terminal destino, aunque a veces también con acciones (como cargar y descargar conteiners). Esta distancia se actualiza con un mensaje en buque que informa sobre su posición actual, y hace que el buque calcule la distancia con la terminal en base a su nueva posición. Para la representación en nuestro trabajo (en perspectiva de una terminal), un buque siempre inicia en un estado Outbound.

El flujo de fases se da como indicado en el enunciado:

Outbound -> Inbound -> Arrived -> Working -> Departing -> Outbound

Roles:

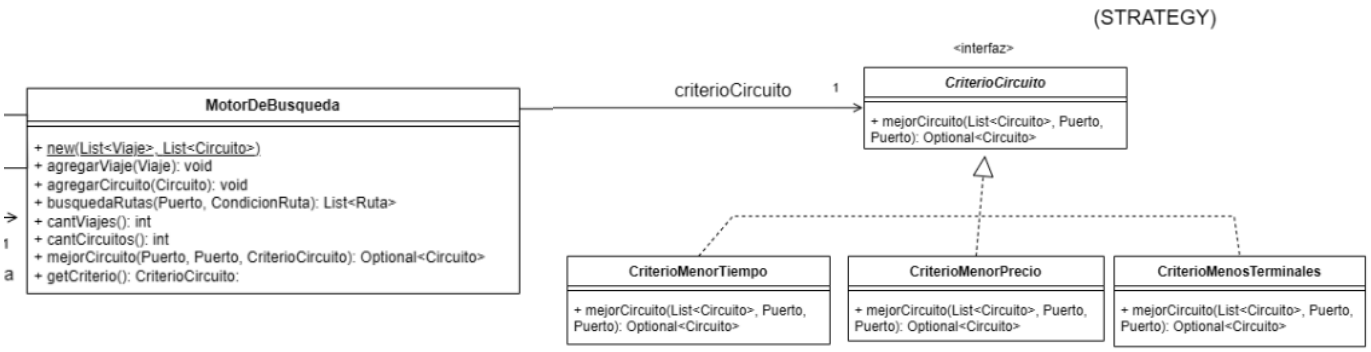
Contexto (Buque): Define la interfaz para el cliente, y conoce una instancia de una clase concreta del estado actual (Fase)

Estado (FaseBuque): Define la interfaz que encapsula el comportamiento que las fases esperan recibir.

Subclase De EstadoContreto(Las 5 fases): Implementa el comportamiento asociado a una fase del buque.

Patrón Strategy en Criterios de Circuito:

Una de las funcionalidades anteriormente mencionadas fue la de conseguir el mejor circuito en base a uno o varios criterios, considerando que podrían agregarse más en un futuro. Para esta funcionalidad implementamos una solución con patrón Strategy.

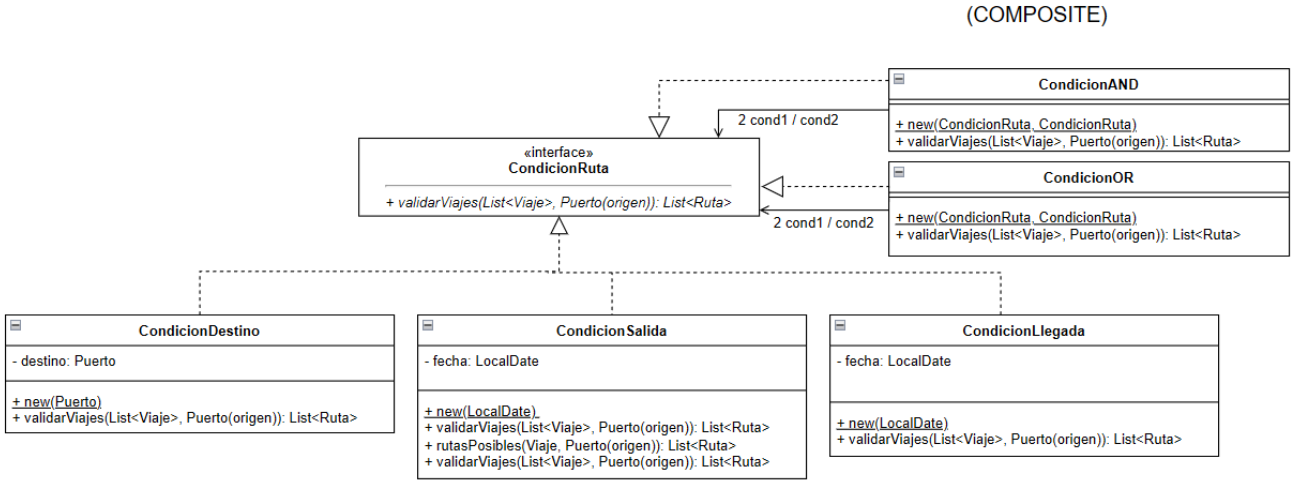


Un motor de búsqueda almacena y conoce su estrategia, la cual además recibe por parámetro cada vez que se le pide buscar el “mejor circuito”. A esta estrategia se le pasan los circuitos que corresponden al trayecto especificado, condición que todos los criterios de circuito deben cumplir, y se le delega la lógica para elegir la mejor opción. Para agregar otra estrategia a futuro bastara con crear la clase concreta implementando la interfaz.

Roles especificados en la página 5 ☺ (funcionalidad 3)

Patrón Composite en Condiciones de Rutas:

La búsqueda de rutas marítimas plantea poder encontrar una o varias rutas que cumplan una o varias condiciones, siempre cumpliendo la de pasar por la terminal gestionada como origen, y usando conectores lógicos AND y OR. Para esta parte implementamos una solución usando el patrón **Composite,** manejando a los casos de conectores como compuestos, y a las condiciones singulares como hojas.



Se considera a una ruta un recorrido desde un puerto hasta otro, dentro de un viaje, y teniendo en cuenta que debe pasarse por el puerto origen antes del destino. Consideramos además que una condición de salida es en relación a la fecha de llegada al origen, ya que un buque llega en una fecha a un puerto, y sale del mismo en la misma fecha.

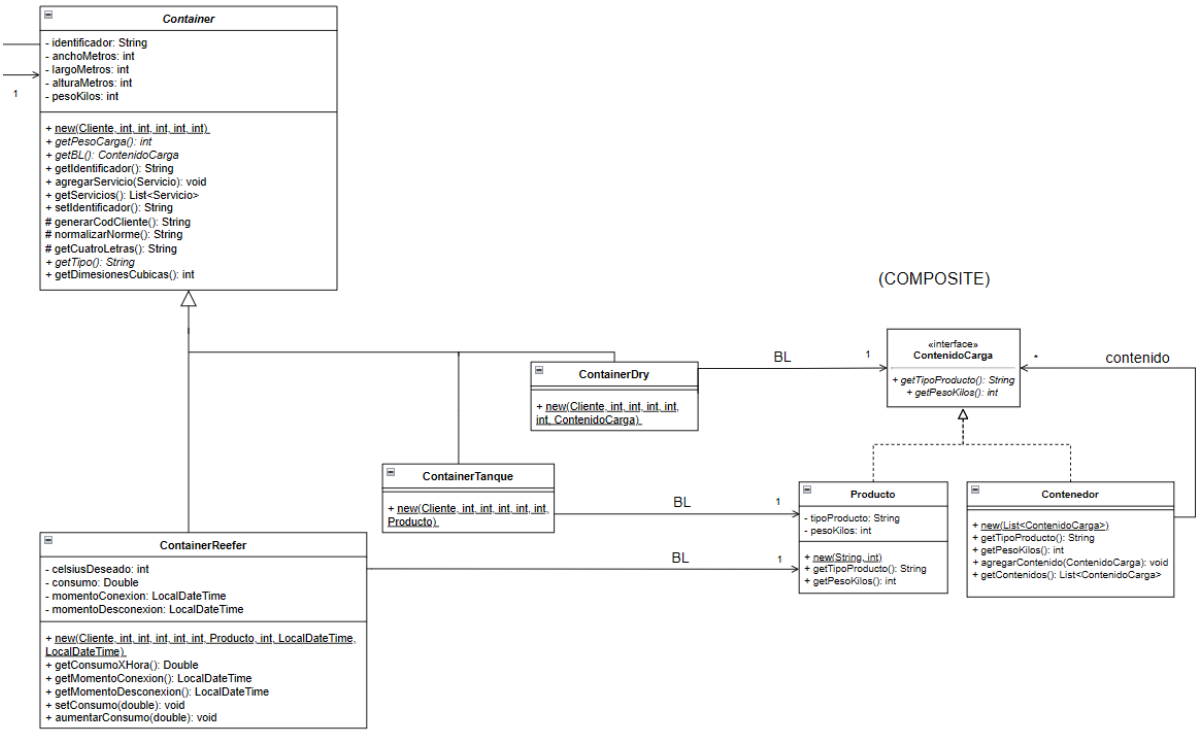
Roles:

Componente(CondicionRuta): Define el contrato para que todas las clases concretas tengan que conseguir rutas a partir de una colección de viajes y un origen.

Hoja(CondicionDestino-Salida-Llegada): Cada clase concreta define su lógica para extraer rutas validas de los viajes dados.

Compuesto (CondicionAND, CondicionOR): And, Mantiene solo las rutas que cumplan ambas de sus condicionesRuta. Or mantiene las rutas que cumplan una u otra condición.

Patrón Composite en BL de Conteiners:



Todo conteiner registra un BL (Bill of Lading) de la carga que lleva dentro. Los conteiners Reefer y Tanque solo pueden llevar una carga sola (Producto), mientras que un conteiner Dry puede llevar un tipo de carga que contenga a otras cargas. Para esto aplicamos el patrón **Composite** nuevamente. Como se ve en la imagen, para representar esta restricción era necesario hacer que las clases concretas definan las relaciones con sus tipos.

Roles:

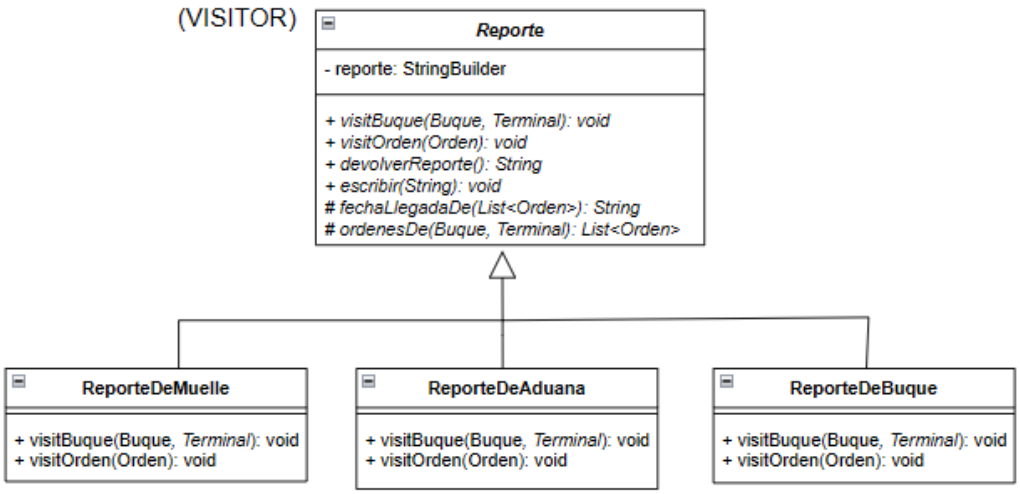
Componente (ContenidoCarga): Define el contrato para que todas las clases concretas tengan que devolver valores como su peso total o su tipo.

Hoja (Producto): Define el comportamiento de los productos, como obtener el peso y valor.

Compuesto (Contenedor): Puede almacenar varios Productos dentro de él.

Patrón Visitor de Reportes:

Para la creación de reportes implementamos una solución utilizando el patrón **Visitor.** La terminal gestionada implementa un método que permite indicar, por parámetro, el buque del cual se busca generar el reporte, y el tipo de formato de reporte que se busca generar. A través de la terminal, dicho buque será llamado para aceptar la visita del reporte, el cual recompilara toda la información necesaria a través de visitas al buque y las ordenes, y armara un String que represente dicho reporte.



Como mencionamos anteriormente, un reporte se crea en base a un buque y una terminal con la que interactúa. Si dicho buque no guarda ninguna orden de exportación/importación que corresponda a dicha terminal, entonces será un reporte vacío (varía según el tipo).

Roles:

Visitor (Reporte): Define el contrato para los visitors concretos.

VisitorConcreto(ReporteMuelle-Aduana-Buque): Implementan los métodos heredados, y definen distintas formas de escribir el reporte y distintos intereses de información para recopilar.

ElementoConcreto(Buque-Orden): El Visitor recopila información atraves de un método “accept”. Nuestro diseño permite que solo el buque necesite entender ese mensaje.

Patrón Template en Reportes:

```
//patron template si no coincide el camion o chofer no se hace nada
public final void manejarLlegada(Terminal term, Camion cam) {
    if (this.condicionTransporte(cam)) {
        this.accionContainer(term);
        this.accionHook();
    }
}

private boolean condicionTransporte(Camion cam) {
    return this.esMismoCamionQueEnOrden(cam) && this.esMismoChoferQueEnOrden(cam);
}
```

Para manejar la llegada de un camión ambas órdenes realizan acciones distintas, pero siguiendo una misma estructura. Para esta implementación utilizamos el patrón Template. Ambas órdenes primero validan la condición de que el camión pueda pasar a la terminal, luego realizan una acción con el container, y finalmente la orden de importación agrega el servicio de almacenamiento excedente en caso de que haya habido un atraso de +24 horas (método hook).

Roles:

ClaseAbstracta (Orden): Implementa el método de plantilla manejarLlegada() y declara operaciones primitivas abstractas y operaciones concretas que son aplicadas en el método plantilla.

ClasesConcretas (Exportación-Importación): implementan las operaciones primitivas, y en caso de usarlos, los métodos de enganche.

CondicionTransporte(): Operación concreta de la clase abstracta, esta condición es siempre igual.

accionContainer(): Operación primitiva implementada por cada clase concreta, las órdenes de exportación almacenan el container. Las órdenes de importación remueven el container.

accionHook(): Método de enganche, solo la orden de importación le da comportamiento. Con este método añade, si corresponde, el servicio de almacenamiento excedente.

Conclusión final

Una vez terminado el trabajo reconocemos el desafío que se nos presentó. Realizando reiteradas e incontables lecturas del enunciado, discutiendo ambigüedades y teniendo diferentes opiniones para resolver un mismo problema, pasando por los diagramas UML, discutiendo interfaces o clases abstractas, detectando patrones de diseño. Todo esto con consideraciones de los diferentes trabajos que se hicieron durante la cursada, la bibliografía de Gamma, papers de SOLID, documentación de tests unitarios con Mockito, y más discusiones.

Como lo fuimos realizando:

Comenzamos por el diseño como aprendimos en la cursada, empezando en una primera instancia a trabajar el diagrama en equipo, para identificar las clases principales y tener un panorama amplio de todo el escenario. Luego nos dividimos los temas, para poder empezar a realizar las tareas en paralelo. Nos pusimos como objetivo no hacer ninguna línea de código hasta no tener todo el UML completo. Decidimos estar abocados solamente a diseñar, y hacer el mayor esfuerzo en esta etapa, a sabiendas de que si lo hacíamos bien, nos iba a traer muchas ventajas para la siguiente etapa, la implementación. Tuvimos un pequeño problema cuando aparecieron los reportes, principalmente por las implicaciones que traía y que ya habíamos descartado con el enunciado anterior. Pero con la recomendación del patrón Visitor, encontramos la forma de extender lo que ya teníamos planeado hacer, sin modificar casi nada.

Con lo aprendido en la materia, sabíamos que si el diseño estaba bien hecho la transición al código iba a ser clara y transparente, y que, si teníamos que agregar, eliminar, o modificar cosas cuando estábamos implementándolo, éstas iban a ser menores y sencillas. Efectivamente, del tiempo total que le dedicamos al trabajo, un 80% fue dedicado al diseño del UML, y el tiempo restante fue para su implementación, y no menos importante, para realizar la documentación del TP. Resaltando nuevamente que la mayor parte del tiempo se destinó al diseño, tuvimos una implementación rápida, sin mayores inconvenientes, y a medida que aparecían nuevas discusiones, las aclarábamos viendo el UML, donde se veía todo de manera más general. Y como era de esperar hubo que agregar, eliminar y modificar cosas, que luego las reflejábamos en el UML.

Palabras finales:

Fue un magnífico trabajo en equipo, cumplimos los requisitos en tiempo y forma. Estamos muy contentos con todo lo realizado y el resultado final obtenido. Este trabajo fue un gran paso en nuestro recorrido de aprendizaje, y nos dio una nueva perspectiva sobre lo que implica diseñar, debatir y aplicar una solución en equipo.