TP Integrador Terminal Portuaria

Integrantes	
Nombre	Correo electrónico
Massa Fabrizio Lautaro	massafabrizio123@gmail.com
Zarco, Julian	julianzarco.2003@outlook.com
Salles Patricio	spatriciojesus@gmail.com

Decisiones de diseño y Detalles de implementación

Buque, GestorDeCoordenadas y Estados

Para la creación de los buques, decidí que debería de conocer a un objeto de GestorDeCoordenadas, el cual tendría la responsabilidad de tener la coordenada en donde el buque está situado actualmente. Un GestorDeCoordenadas se encarga de hacer comparaciones de distancia entre el buque y la terminal de destino u origen (*En el caso de hacer departing*); Con esto le permite saber al buque que tan cerca está de una terminal y a que estado pasar depende de la situación (*Ejemplo: Inicia en Outbound, pero cuando está a menos de 50km de distancia, pasa a Inbound y avisa a los importadores del viaje actual, cuando está a 0 de distancia, entra en Arrived y también avisa a los importadores. Por último, si está en Depart y se va más de 1km de la terminal origen del tramo actual, ya entra en outbound hacia la otra terminal destino que tiene en su tramo si es que tiene que seguir en los tramos del Viaje y avisa a los exportadores del viaje*).
También el Buque debe de conocer a un Viaje (*Opcionalmente*), el cual le da la información de los tramos que debe de tomar y cual es la terminal destino y origen actual.
Cuando un buque pasa de outbound a inbound, le manda un mensaje a la terminal destino del viaje para que le avise a los importadores según el viaje que le pertenece y cuando sale de la terminal destino, le avise a los exportadores del viaje.
Un buque cuando se sitúa en Arrived, puede pasar a Working, donde debe de esperar al aviso de la terminal para partir y cambiar su estado a Depart (*En esta fase estaría descargando y cargando desde la terminal ubicada*), una vez ordenado de retirarse, le indica al viaje que debe actualizar su tramo y indicar cuales son los nuevos terminal destino y origen.
Los buques se pueden identificar por Nombre y al instanciar inician en Outbound.
Para el buque tuve que usar el patrón de State para poder manejar los distintos estados del buque según su distancia con la terminal. En total tuve que diseñar 5 estados, según lo que pedía el dominio.

Lineas navieras, circuitos y viajes

En primer lugar, cada circuito de una naviera está formado por un ArrayList de Tramos. Cada objeto de clase Tramo indica la terminal de origen y de destino donde empieza, al igual que el tiempo que toma recorrer dicho tramo y el precio monetario que cuesta dicho recorrido. A un tramo también se le puede preguntar si inicia o termina en una determinada Terminal.

Dentro de la clase Circuito, declaré un método privado llamado “ordenarTramos” que, dado una lista de Tramos, devuelve una nueva lista con todos los tramos ordenados de tal forma que los tramos formen un circuito cíclico cerrado. Esto se hizo para evitar que un Circuito no cumpla con la condición de tener sus tramos ordenados, y que el destino del último tramo conecte con el inicio del primer tramo.
Adicionalmente, dado una terminal de origen y otra de destino, a un circuito se le puede pedir que forme un nuevo Viaje.

La clase Viaje tiene como atributos a una terminal portuaria de origen, una terminal de destino, el circuito del que forma parte el recorrido del viaje, la fecha de salida del viaje desde su origen, y un par de variables: una indica la última terminal de la que partió un buque, y la otra indica la próxima terminal a la que debe ir un buque.
Esos dos últimos atributos cambian dinámicamente cuando un buque llega a una de las terminales por las que debe pasar, mientras que los tres primeros atributos no cambian nunca y son fijos.
Luego, un viaje puede responder a consultas que otras clases hagan gracias a un conjunto de métodos públicos definidos dentro de la clase: se le puede pedir el tiempo de transcurso que conlleva pasar por todos los tramos del viaje; la fecha de llegada considerando el tiempo de transcurso y la fecha de salida; el precio total que implica el recorrido de cada uno de sus tramos; todas las terminales portuarias que están dentro del viaje; el puerto de llegada del último tramo, que equivale al destino final del viaje; y por último, una instancia de Viaje puede verificar si una terminal dada existe dentro de su recorrido.

Luego, cada línea naviera tiene tres listas: una para sus buques, una para sus circuitos, y la última para registrar los recorridos que deben hacer las naves.
Para la última, decidí encapsular la lógica de los recorridos dentro de una clase Wrapper, la cual contiene un buque y un ArrayList de Viajes que el buque debe recorrer.
Dados un buque y una lista de viajes, el usuario puede agregar nuevas instancias de WrapperRecorrido a la naviera; siempre y cuando el buque dado se encuentre dentro de la naviera, caso contrario se devuelve un error.

Por último, dada una terminal portuaria, la línea naviera puede devolver una lista con los circuitos que pasan por dicha terminal. Esto fue especialmente diseñado para que la Terminal Gestionada averigüe si agregar una naviera a su colección o no; si dicha terminal no es cubierta por al menos un circuito de la naviera, no se va a agregar a la lista de la terminal.

Mejor circuito

La Terminal Gestionada cuenta con un sistema para encontrar un circuito óptimo que lo conecta con cualquier otra terminal dada. Para esto, se creó una clase abstracta llamada “E_MejorRuta”.

Esta clase cuenta con un método protegido llamado “circuitosQueContienen”, que toma los circuitos de las líneas navieras que contienen a la Terminal Gestionada y filtra todos sus circuitos, quedando solamente aquellos que conectan a la Terminal Gestionada con una terminal de destino a elección del usuario.
Luego, la clase “viajesPorCadaCircuito” forma un Map donde las claves son los viajes que conectan a la Terminal Gestionada con la terminal destino, y el valor asociado a la clave son los circuitos que se tomarán para hacer dichos viajes.
Finalmente, el método “mejorCircuitoHacia” toma el Map del método anterior y, mediante un método abstracto llamado “criterioDeSeleccion” que es redefinido por todas las subclases de E_MejorRuta, encuentra finalmente el circuito óptimo para realizar el recorrido.

En el modelo actual, hay tres posibles conceptos de “circuito óptimo”; sin embargo, el modelo queda abierto a posibles nuevos conceptos en el futuro. Los tres conceptos actualmente son:

- El circuito cuyo viaje desde la Terminal Gestionada hasta la terminal destino cueste menos dinero.
- El circuito cuyo viaje toma menos tiempo en completar.
- El circuito con el menor número de terminales entre el origen y el destino.

Containers, Bill Of Landing y Productos

Para los containers, cree una clase padre abstracta llamada Container.
Cada Container posee un ancho, largo, altura, identificador y una Interfaz de BillOfLanding (BL) la cual permite decidir diferentes tipos de diseño de reconocimiento de BLs para futuros contenedores (*Lo que les va a permitir usar el modelo de BLs compuestos o solo conocer a 1 solo BL, o lo que sea que se implemente a futuro en relación a los BLs*).
Los containers pueden preguntar por los dueños / dueño según el Bill of landing asociado al container, obtener los bill of landing si es que es compuesto o obtener el bill of landing si no lo es, obtener el peso, agregar bill of landings si es un container compuesto de otros bill of landing, calcular la capacidad y calcular el costo de consumo eléctrico en kw/h si es un Reefer.
Para los containers que pueden componerse de otros BLs, pueden decidir implementar un Bill of landing especial debido a que la interfaz le permite adaptar cualquier otros tipos de BL que existan.
Los Bill of Landing se componen de un dueño (*O varios en el caso del especial*) y una lista de productos, de cada producto se sabe su peso y nombre.
El bill of landing sabe calcular el peso total de la suma de todos los pesos de los productos listados. También puede describir qué productos posee, su dueño o dueños, agregar más bl en caso de ser especial y describirse o describir sus bls.
Para los containers no necesite ningún patrón en particular, pero si el patrón Composite para Bill of Landing, debido a la existencia de BLs especiales que pueden componerse de otros BLs.

Para esto tuve que implementar asserts para asegurar que no se introduzcan datos erróneos con respecto a la identificación de cada contenedor, ancho, largo y altura e incluso el consumo eléctrico para los Reefer.
Algo particular que tuve que implementar para los dry container, fue que su identificador es aleatoriamente creado, debido a que en este caso con la posibilidad de componerse de varios BLs de diferentes clientes, era mejor darle un nombre por default (*Siendo “DRYC”*) y dejar que un algoritmo decida su identificador en dígitos aleatoriamente, por lo tanto para este caso no se requiere ingresar un identificador.

Para poder calcular el peso y las propiedades de los productos que lleva el Bill of Landing, decidí crear un Value Object llamado Producto, el cual puedes asignarle un nombre y un peso. El bill of landing calculara el peso total al sumar el peso de todos sus productos vinculados en el.

Servicios

Los servicios fueron hechos usando una clase abstracta de Servicio, la cual permite definir un servicio de diferentes formas: Solo teniendo un container o con un container y un precio fijo. Cada servicio puede definir a su manera qué datos deberá de incluir en sus campos adicionales.

Como protocolo, los servicios pueden responder a mensajes para calcular su precio (*Si es que el servicio tiene permitido*), Describir un cliente en específico para el caso de la Desconsolidación y verificar si hay pérdidas en un tanque.
Tuve que diseñar servicios adicionales que fueron mencionados en algunas partes del Dominio, como la desconsolidación (*Para los casos en donde un Dry Container se compone de un BL Especial*) y la revisión de pérdidas para Tank Containers.
El resto de los servicios de contenedores agregados fueron Almacenamiento Excedente, Electricidad (*Para Containers Reefers*), Servicio de pesado y Lavado (*El cual permite definir cuál será el precio mayor y precio menor según la capacidad del container*),
Para esta parte, no tuve ningún patrón en mente.
Tuve que implementar asserts para verificar que los datos sean correctos y no se admitan datos erróneos, como un precio negativo.

Reportes

Con el propósito de informar a los participantes de la operatoria de una terminal, se crearon las interfaces “ReporteVisitor” y “ElementoVisitable”.

El primero define cuatro métodos, tres de los cuales son métodos “visitar” que toman instancias de clases distintas (TerminalPortuaria, OrdenImportacion, OrdenExportacion), al igual que un buque, como parámetros. Las tres clases que implementan sus métodos son:

- “ReporteMuelleVisitor”, que genera un reporte de un buque que arribó a la terminal e informa la fecha de llegada, fecha de partida y el número de contenedores operados durante su tiempo en la terminal.
- “ReporteAduanaVisitor”, que indica el nombre de un buque arribado, fecha de llegada y salida, e información sobre los containers que llevaba.
- “ReporteBuqueVisitor”, que muestran los identificadores de los contenedores que fueron cargados y descargados del buque.

Posteriormente, la interfaz “ElementoVisible” ofrece un método “aceptar”, y es implementada por las clases TerminalPortuaria, OrdenImportacion y OrdenExportacion. Todas estas clases son visitadas por las tres clases de ReporteVisitor y, dependiendo de la clase que visiten, los reportes se formarán de unas u otras formas.

Por último, la clase TerminalPortuaria cuenta con un método “generarReporteDeBuque” que, dado una clase de ReporteVisitor y un buque, se genera un reporte del tipo dado relacionado al buque seleccionado.

Patrones de diseño utilizados

Órden de exportación: Strategy

Con el objetivo de que la terminal busque y encuentre el mejor circuito para llevar a cabo una orden de exportación, se utilizó el patrón Strategy.

Roles:

Estrategia: E_MejorRuta. Esta es la clase abstracta que define algunos métodos protegidos, al igual que un método abstracto que las subclases deberán rellenar a sus maneras.

```
public abstract class E_MejorRuta {
    public E_MejorRuta() {}

    public Circuito mejorCircuitoHacia(TerminalPortuaria terminalOrigen, TerminalPortuaria terminalDestino) {
        final HashMap<Viaje,Circuito> mapa = viajesPorCadaCircuito(terminalOrigen, terminalDestino);

        var viajeOptimo = this.criterioDeSeleccion(mapa.keySet());

        return mapa.get(viajeOptimo);
    }

    protected abstract Viaje criterioDeSeleccion(Set<Viaje> mapa);

    protected List<Circuito> circuitosQueContienen(TerminalPortuaria terminalOrigen, TerminalPortuaria terminalDestino) {
        return terminalOrigen.getMisNavieras().stream()
                                        .flatMap(nav -> nav.circuitosQuePasanPor(terminalDestino).stream())
                                        .toList();
    }

    protected HashMap<Viaje, Circuito> viajesPorCadaCircuito(TerminalPortuaria terminalOrigen, TerminalPortuaria terminalDestino) {
        HashMap<Viaje, Circuito> mapCircuitoConViaje = new HashMap<Viaje, Circuito>();

        for (Circuito circ : this.circuitosQueContienen(terminalOrigen, terminalDestino)) {
            mapCircuitoConViaje.put(circ.crearNuevoViaje(terminalOrigen,terminalDestino,LocalDateTime.now()), circ);
        }

        return mapCircuitoConViaje;
    }
}
```

Estrategias concretas: E_MenorPrecio, E_MenorPrecio, E_TerminalesIntermedias. Las clases que heredan de E_MejorRuta y definen sus propios algoritmos de búsqueda de mejor circuito.

```
public class E_MenorPrecio extends E_MejorRuta {

    public E_MenorPrecio() {}

    @Override
    protected Viaje criterioDeSeleccion(Set<Viaje> mapa) {
        return mapa.stream()
                   .min(Comparator.comparing(Viaje::precioTotal))
                   .orElse(null);
    }

}
```

```
import ar.edu.unq.po2.TerminalPortuaria.NavierasYCircuitos.Viaje;

public class E_MenorTiempo extends E_MejorRuta {

    public E_MenorTiempo() {}

    @Override
    protected Viaje criterioDeSeleccion(Set<Viaje> mapa) {
        return mapa.stream()
                   .min(Comparator.comparing(Viaje::duracionDelViaje))
                   .orElse(null);
    }

}
```

```java
public class E_TerminalesIntermedias extends E_MejorRuta {

    public E_TerminalesIntermedias() {}

    @Override
    protected Viaje criterioDeSeleccion(Set<Viaje> mapa) {
        return mapa.stream()
                   .min(Comparator.comparing(Viaje::numeroDeTerminalesEntreOrigenYDestino))
                   .orElse(null);
    }

}
```

Contexto: TerminalPortuaria. La clase hace uso de distintas estrategias concretas. La elección de la estrategia óptima queda a disposición del usuario.

```java
public void setEstrategia( E_MejorRuta estrategia ) {
   this.estrategia = estrategia;
}

public E_MejorRuta getEstrategia() {
    return this.estrategia;
}


public Circuito getMejorCircuito(TerminalPortuaria terminalDestino) {
    return estrategia.mejorCircuitoHacia(this, terminalDestino);
}
```

Reportes: Visitor

Se utilizó el patrón Visitor para modelar el funcionamiento del mandado de reportes desde la terminal. Se eligió este patrón ya que el sistema cuenta con múltiples clases relacionadas con la operatoria portuaria; cada una de ellas posee información distinta, pero a la vez todas participan en la generación de los distintos reportes (Muelle, Aduana, Buque).

Con tal de evitar las clases del modelo con código de formato o presentación, se usó el patrón Visitor.

Roles:

Visitante: ReporteVisitor. Esta clase define la interfaz común para todas las clases ReporteVisitor, declarando tres métodos “visitar” que difieren por el tipo de dato de su primer parámetro.

Si bien no es común pasar un segundo parámetro de otro tipo a un método “visitar” del patrón Visitor, nos pareció el método más práctico y rápido de hacerle saber a las subclases con qué buque se está trabajando.

```java
import ar.edu.unq.po2.TerminalPortuaria.Buque.Buque;
import ar.edu.unq.po2.TerminalPortuaria.Orden.OrdenExportacion;
import ar.edu.unq.po2.TerminalPortuaria.Orden.OrdenImportacion;
import ar.edu.unq.po2.TerminalPortuaria.Terminal.TerminalPortuaria;

public interface ReporteVisitor {
    void visitar(TerminalPortuaria terminal, Buque buque);
    void visitar(OrdenImportacion orden, Buque buque);
    void visitar(OrdenExportacion orden, Buque buque);
    String generarReporte();
}
```

Visitantes concretos: ReporteAduanaVisitor, ReporteBuqueVisitor, ReporteMuelleVisitor. Estas tres clases implementan la lógica concreta de cada tipo de reporte, aplicando comportamientos diferentes sobre los mismos elementos.

```java
public class ReporteAduanaVisitor implements ReporteVisitor {
    private StringBuilder html = new StringBuilder();

    @Override
    public void visitar(TerminalPortuaria terminal, Buque buque) {
        html.append("<html><body>");
        html.append("<h2>Reporte de Aduana</h2>");
        html.append("<p>Buque: ").append(buque.getNombreBuque()).append("</p>");
        html.append("<ul>");
    }

    @Override
    public void visitar(OrdenImportacion orden, Buque buque) {
        html.append("<li>Container tipo: IMPORTACIÓN - ID: ")
            .append(orden.getBill().getBillOfLandings().get(0).hashCode())
            .append("</li>");
    }

    @Override
    public void visitar(OrdenExportacion orden, Buque buque) {
        html.append("<li>Container tipo: EXPORTACIÓN - ID: ")
            .append(orden.getBill().getBillOfLandings().get(0).hashCode())
            .append("</li>");
    }

    public String generarReporte() {
        html.append("</ul></body></html>");
        return html.toString();
    }
}
```

```java
public class ReporteBuqueVisitor implements ReporteVisitor {
    private StringBuilder xml = new StringBuilder();

    public ReporteBuqueVisitor() {
        xml.append("<report>\n<import>\n");
    }

    @Override
    public void visitar(TerminalPortuaria terminal, Buque buque) {} // Nada en particular con la terminal

    @Override
    public void visitar(OrdenImportacion orden, Buque buque) {
        xml.append("<item>")
            .append(orden.getBill().getBillOfLandings().get(0).hashCode())
            .append("</item>\n");
    }

    @Override
    public void visitar(OrdenExportacion orden, Buque buque) {
        xml.append("</import>\n<export>\n");
        xml.append("<item>")
            .append(orden.getBill().getBillOfLandings().get(0).hashCode())
            .append("</item>\n");
    }

    public String generarReporte() {
        xml.append("</export>\n</report>");
        return xml.toString();
    }
}
```

```java
public class ReporteMuelleVisitor implements ReporteVisitor {
    private StringBuilder reporte = new StringBuilder();
    private int contenedoresOperados = 0;
    private LocalDateTime arribo;
    private LocalDateTime partida;

    @Override
    public void visitar(TerminalPortuaria terminal, Buque buque) {
        reporte.append("=== Reporte Muelle ===\n");
        reporte.append("Buque: ").append(buque.getNombreBuque()).append("\n");
        arribo = buque.getViaje().getFechaSalida();  // o como corresponda
        partida = buque.getViaje().fechaDeLlegada();
    }

    @Override
    public void visitar(OrdenImportacion orden, Buque buque) {
        contenedoresOperados += 1;
    }

    @Override
    public void visitar(OrdenExportacion orden, Buque buque) {
        contenedoresOperados += 1;
    }

    public String generarReporte() {
        reporte.append("Fecha arribo: ").append(arribo).append("\n");
        reporte.append("Fecha partida: ").append(partida).append("\n");
        reporte.append("Contenedores operados: ").append(contenedoresOperados).append("\n");
        return reporte.toString();
    }
}
```

Elemento: ElementoVisitable. La interfaz con el método “aceptar” que luego implementarían otras clases.

```java
package ar.edu.unq.po2.TerminalPortuaria.Reportes;

import ar.edu.unq.po2.TerminalPortuaria.Buque.Buque;

public interface ElementoVisitable {
    void aceptar(ReporteVisitor visitor, Buque buque);
}
```

Elementos concretos: TerminalPortuaria, OrdenImportacion, OrdenExportacion. Estas tres clases son las que implementan “aceptar” de la interfaz ElementoVisitable y exponen sus datos a los visitantes mediante su método.

```java
@Override
public void aceptar(ReporteVisitor visitor, Buque buque) {
    visitor.visitar(this, buque);

    for (Orden orden : ordenes) {
        orden.aceptar(visitor, buque);
    }
}
```

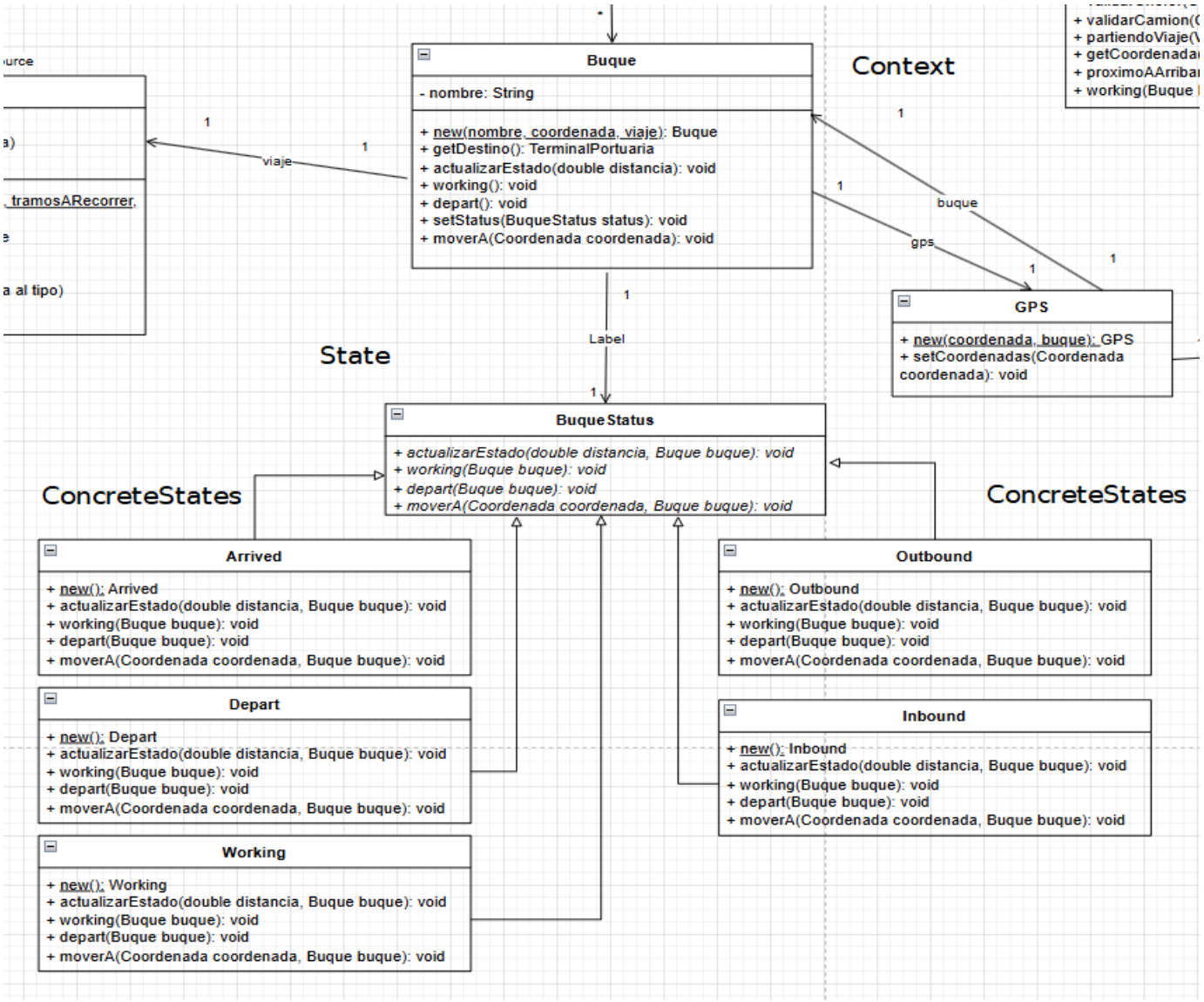
```java
@Override
public void aceptar(ReporteVisitor visitor, Buque buque) {
    visitor.visitar(this, buque);
}
```

Estructura de Objeto: TerminalPortuaria. En este modelo, TerminalPortuaria actúa tanto de elemento concreto como de estructura de objeto, porque además de ser visitable, administra y recorre la colección de elementos visitables (las órdenes).

```java
public String generarReporteDeBuque(ReporteVisitor visitor, Buque buque) {
    this.aceptar(visitor, buque);
    return visitor.generarReporte();
}
```

Buque: State

Para poder lograr que el buque reaccione de manera diferente según la coordenada en la que se sitúa y su distancia con la terminal. Decidí usar este patrón. En total se compone de 5 estados: *Outbound (Estado inicial), Inbound, Arrived, Working, Depart*.



En el estado de **Outbound** puede:

- Moverse a otra Coordenada.
- Verificar que tan cerca está de la terminal destino, si está a menos de 50km pasa a Inbound y manda un mensaje a los importadores, en el caso de estar a 0km pasa a Arrived y le avisa a los importadores.

Pero no se va a poder demandarle que pase a estar en Working ni Depart.

En el estado de **Inbound** puede:

- Moverse a otra Coordenada.
- Verificar que tan cerca está de la terminal destino, si está a 0km de distancia pasa a Arrived, pero en caso de alejarse a más de 50km pasa a Outbound de vuelta.

Pero no se va a poder demandarle que pase a estar en Working ni Depart.

En el estado de **Arrived** puede:

- Pasar a Working.

No hace nada pedirle que actualice su estado según la distancia a la terminal, ya que no debería poder moverse en este estado.
No puede moverse ni pasar a Depart.

En el estado de **Working** puede:

- Realizar operaciones de descarga y carga.
- Actualizar su tramo a su siguiente en el Viaje y pasar a Depart.

No puede moverse y actualizar su estado hace nada.

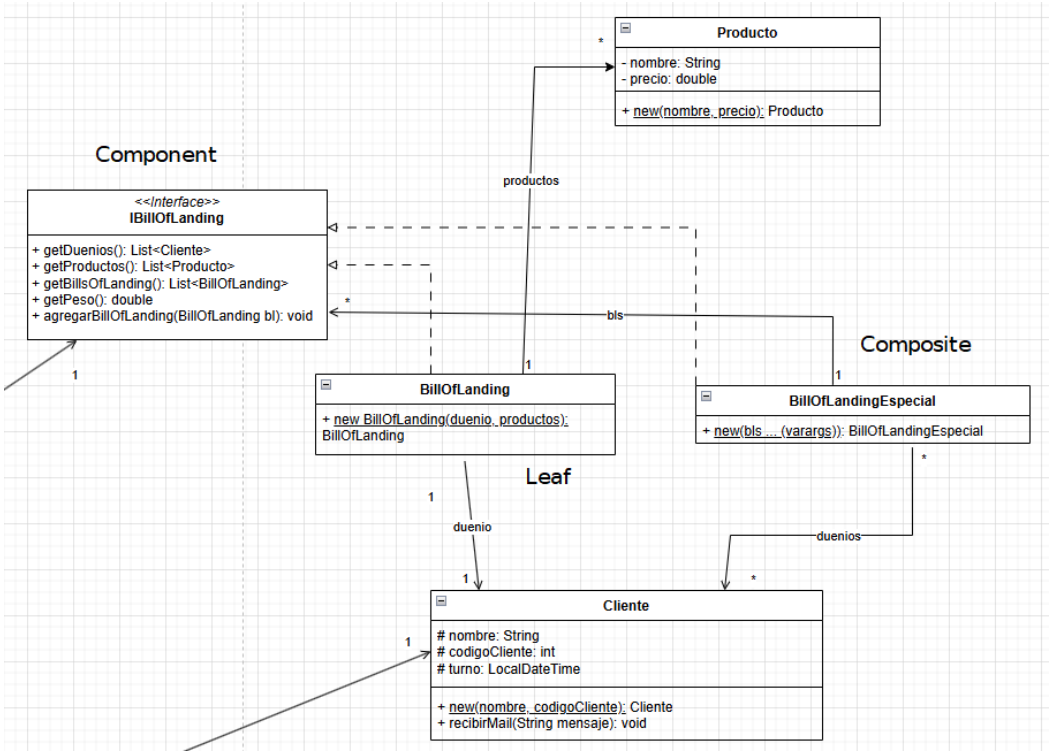
En el estado Depart **puede**:

- Moverse a otra Coordenada.
- Verificar que tan lejos está de la terminal origen, si ya se fue más de 1km, entonces debe de avisar a los exportadores y volver a outbound para dirigirse a la siguiente terminal de destino, teniendo en cuenta su siguiente terminal destino en el tramo o terminar el Viaje si es que ya no hay más tramos a cubrir.

Solo no se le puede pedir que haga Working, porque ya partió de la terminal.

Bill Of Landing: Compose

Elegí el patrón de compose para esta sección, debido a que existe la posibilidad de un Bill of Landing que se compone de otros BLs. Este patrón me pareció esencial para representar este comportamiento.
Por lo tanto decidí la siguiente estructura:



El Bill of Landing especial se puede componer de otros BL siendo que es el Composite, mientras que el Bill of landing normal no puede ya que es el Leaf.
He decido tomar la alternativa de tener transparencia sobre la interfaz Component para que otros clientes de esta interfaz puedan manejar los mensajes para agregar BLs a un BL Especial, con el fin de evitar posibles casting de clases de instancias.